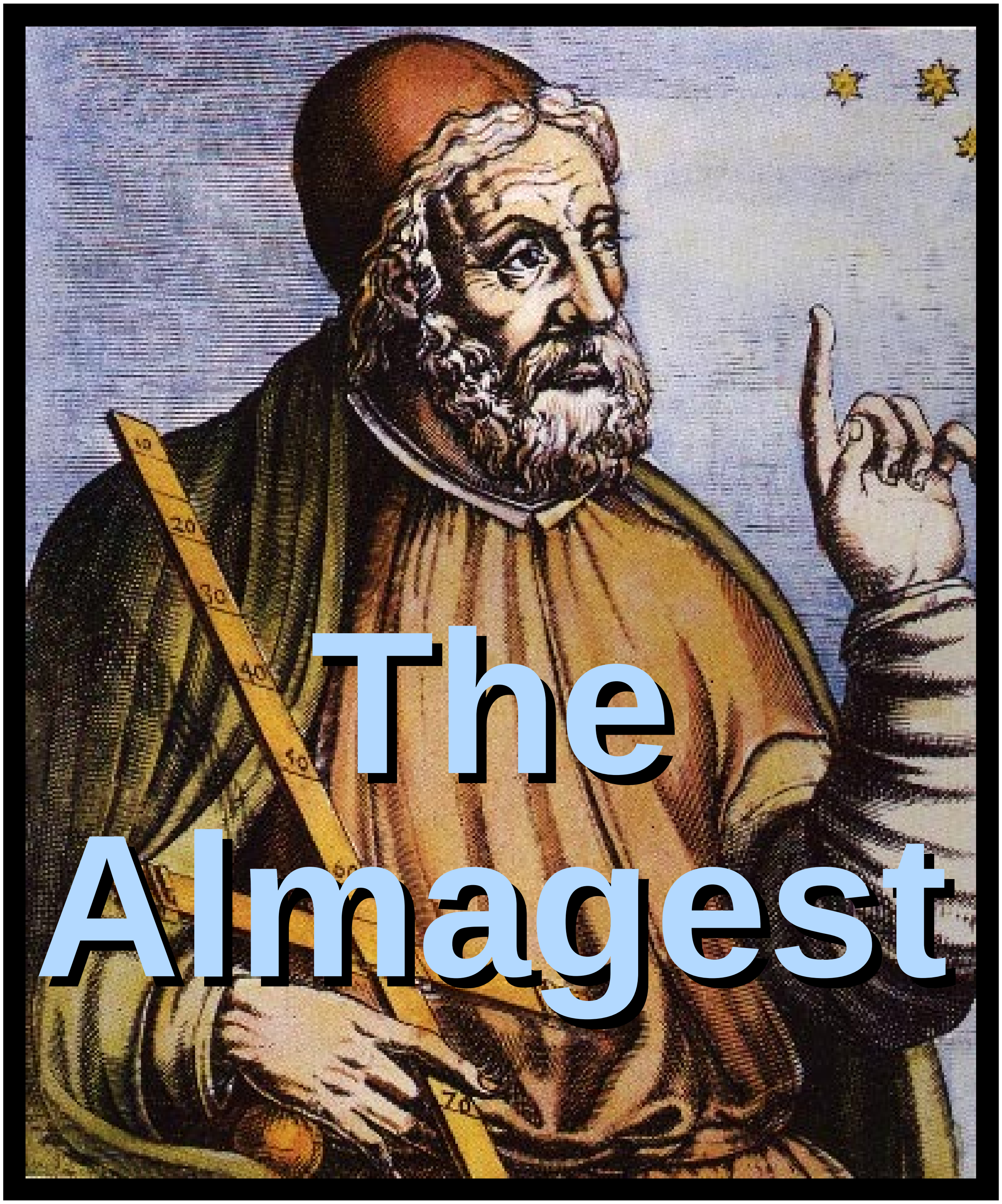
UNIVERSITY OF CALIFORNIA AT BERKELEY

COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES
BERKELEY, CALIFORNIA 94720



The Almagest

Vol. 1 - Ptolemy 0.7 User's Manual

Primary Authors

Shuvra Bhattacharyya, Joseph T. Buck, Wan-Teh Chang, Michael J. Chen, Brian L. Evans, Edwin E. Goei, Soonhoi Ha, Paul Haskell, Chih-Tsung Huang, Wei-Jen Huang, Christopher Hylands, Asawaree Kalavade, Alan Kamas, Allen Lao, Edward A. Lee, Seungjun Lee, David G. Messerschmitt, Praveen Murthy, Thomas M. Parks, José Luis Pino, John Reekie, Gilbert Sih, S. Sriram, Mary P. Stewart, Michael C. Williamson, Kennard White.

Other contributors

Raza Ahmed, Egbert Amicht (AT&T), Sunil Bhave, Anindo Banerjea, Neal Becker (Comsat), Jeff Bier, Philip Bitar, Rachel Bowers, Andrea Cassotto, Gyorgy Csertan (T.U. Budapest), Stefan De Troch (IMEC), Rolando Diesta, Martha Fratt, Mike Grimwood, Luis Gutierrez, Eric Guntvedt, Erick Hamilton, Richard Han, David Harrison, Holly Heine, Wai-Hung Ho, John Hoch, Sangjin Hong, Steve How, Alireza Khazeni, Ed Knightly, Christian Kratzer (U. Stuttgart), Ichiro Kuroda (NEC), Tom Lane (Structured Software Systems, Inc.), Phil Lapsley, Bilung Lee, Jonathan Lee, Wei-Yi Li, Yu Kee Lim, Brian Mountford, Douglas Niehaus (Univ. of Kansas), Maureen O'Reilly, Sunil Samel (IMEC), Chris Scannel (NRL), Sun-Inn Shih, Mario Jorge Silva, Rick Spickelmier, Eduardo N. Spring, Richard S. Stevens (NRL), Richard Tobias (White Eagle Systems Technology, Inc.), Alberto Vignani (Fiat), Gregory Walter, Xavier Warzee (Thomson), Anders Wass, Jürgen Weiss (U. Stuttgart), Andria Wong, Anthony Wong, Mei Xiao, Chris Yu (NRL).

Current Sponsors

Various parts of the Ptolemy project have been supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the Semiconductor Research Corporation (SRC) (project 95-DC-324-016), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby Laboratories, Hitachi, Mentor Graphics, Mitsubishi, Motorola, NEC, Pacific Bell, Philips, and Rockwell.

The Ptolemy project is an ongoing research project focusing on design methodology for heterogeneous systems. Additional support for further research is always welcome.

Trademarks

About the Cover

The image on the cover is from an engraving at the Granger Collection in New York. It depicts Claudius Ptolemy, an astronomer from the second century A. D. Ptolemy codified the Greek geocentric view of the universe, and rationalized the apparent retrograde motion of the planets using *epicycles*. The Ptolemaic system remained the accepted wisdom until the Polish scholar Copernicus proposed a heliocentric view in 1543.

Contents

Chapter 1. An Overview of Ptolemy

1.1 Introduction

The core of Ptolemy is a compact software infrastructure upon which specialized design environments (called *domains*) can be built. The software infrastructure, called *the Ptolemy kernel*, is made up of a family of C++ class definitions. Domains are defined by creating new C++ classes derived from the base classes in the kernel.

Domains can operate in either of two modes:

- Simulation — A scheduler invokes code segments in an order appropriate to the model of computation.
- Code generation — Code segments in an arbitrary language are stitched together to produce one or more programs that implement the specified function.

The use of an object-oriented software technology permits a domain to interact with one another without knowledge of the features or semantics of the other domain. Thus, using a variety of domains, a team of designers can model each subsystem of a complex, heterogeneous system in a natural and efficient manner. These different subsystems can be nested to form a tree of subsystems. This hierarchical composition is key in specifying, simulating, and synthesizing complex, heterogeneous systems.

By supporting heterogeneity, Ptolemy provides a research laboratory to test and explore design methodologies that support multiple design styles and implementation technologies. A simple example is simulating the effects of transmitting compressed video and audio over an asynchronous transfer mode (ATM) network. The network will delay, drop, and reorder packets based on the congestion. Compression and decompression, however, work on the video and audio data, and the time associated with the data is not relevant to the signal processing. The simulation in this case is heterogeneous: the network processes discrete events (packets) with a notion of time, whereas the signal processing processes data independent of time. Other examples of heterogeneous systems include integrated control and signal processing architectures, mixed analog/digital simulation, and hardware/software codesign.

In short, Ptolemy is a flexible foundation upon which to build prototyping environments. The Ptolemy 0.7 release contains, for example, dataflow-oriented graphical programming for signal processing [Lee87a,b][Buc91][Buc93a,b,c], a multi-threaded process networks modeling environment [Par95], a synchronous/reactive programming framework [Edw97], discrete-event modeling of communication networks [Wal92][Hal93][Cha97], and synthesis environments for embedded software [Bha93a,b,c][Bha94a,b][Pin95]. We have also developed prototyping environments that are not released with Ptolemy 0.7, such as design assistants for hardware/software codesign [Kal93]. The Ptolemy system is fundamentally extensible, as we release all of the source code. Users can create new component models, new design process managers, and even entirely new programming environments.

1.2 History

Ptolemy is a third-generation software environment that started in January of 1990. It is an outgrowth of two previous generations of design environments, Blosim [Mes84a,b] and Gabriel [Lee89][Bie90], that were aimed at digital signal processing (DSP). Both environments use dataflow semantics with block-diagram syntax for the description of algorithms. To broaden the applicability beyond DSP, the Ptolemy kernel does not build in dataflow semantics, but instead provides support for a wide variety of computational models, such as dataflow, discrete-event processing, communicating sequential processes, computational models based on shared data structures, and finite-state machines. For these computational models, the Ptolemy kernel provides a mixture of compile-time and run-time scheduling techniques. Unlike Blosim or Gabriel, then, the Ptolemy kernel provides infrastructure that is extensible to new computational models without re-implementation of the system.

Since 1990, we have had seven major releases of Ptolemy, numbered 0.1 through 0.7. The zero indicates that Ptolemy is research software and not a commercial product. Between annual major releases, we put out one or two incremental releases. Our goal is to test our algorithms and methodologies in Ptolemy and to transfer them as quickly as possible to the public through freely distributable releases. Because of the critical mass of users of Ptolemy worldwide, a news group called `comp.soft-sys.ptolemy` was formed in 1994. The Ptolemy Web site `http://ptolemy.eecs.berkeley.edu/` went on-line in May of 1994.

The flexibility of Ptolemy is particularly important for enabling research in design methodology. In September of 1993, the Ptolemy project became part of the technology base portion of the RASSP project (rapid prototyping of application-specific signal processors), organized and sponsored by Advanced Research Projects Agency (ARPA) and the United States Air Force. The Ptolemy part of the RASSP project was to research system-level design methodology for embedded signal processors. Our project aimed to develop formal models for such heterogeneous systems, a software environment for the design of such systems, and synthesis technologies for implementation of such systems. In the latter category, we have been concentrating on problems not already well addressed elsewhere, such as the synthesis of embedded software and the partitioning and scheduling of heterogeneous parallel systems. In 1997 the project became part of the DARPA Composite CAD program, and has shifted its focus towards more aggressively heterogeneous systems, including for example microelectromechanical components, and distributed adaptive signal processing systems.

We have transferred many of our research ideas to computer-aided design tool vendors such as Cadence, Hewlett Packard, and Synopsys. Cadence's Signal Processing Workshop (SPW) includes their version of our synchronous dataflow (SDF) domain and multirate dataflow schedulers. Cadence's Convergence environment (released in October, 1995) is Cadence's implementation of our ideas for heterogeneous simulation. Cadence has used Convergence to allow SPW and Bones (a discrete-event simulator) to cooperate in a simulation, just as the Ptolemy kernel has allowed the SDF domain and the Discrete-Event (DE) domain since 1990. Berkeley Design Technology has a similar cosimulation environment for SPW and Bones, but their implementation is based on the Ptolemy kernel. In June of 1997, Hewlett Packard announced plans to release a Ptolemy-based dataflow modeling environment that is integrated with their highly regarded analog, RF, and microwave circuit simulation software.

1.3 Ptolemy Kernel

The overall organization of the latest release of the Ptolemy system is shown in figure 1-1. A typical use of Ptolemy involves starting two Unix^{TM} processes, as shown in figure 1-1(a), by running `pigi` (Ptolemy interactive graphical interface). The first process contains the `vem` user interface and the `oct` design database [Har86], and the other process contains the Ptolemy kernel. An alternative is to run Ptolemy without the graphical user interface, as a single process, as shown in figure 1-1(b). In this case, the textual interpreter is based on the Tool Command Language, Tcl [Ous90][Ous94], and is called call `ptcl` for Ptolemy Tcl. It is possible to design other user interfaces for the system. We are releasing a preliminary version of a third interface called Tycho. In its current form, Tycho is best suited for language-sensitive editing and consoles for tools such as Matlab and Mathematica.

The executable programs `pigiRpc` or `ptcl` can be configured to include any subset of the available domains. The most recent picture of the domains that Berkeley has developed is shown in figure 1-2. Many different styles of design are represented by these domains. More are constantly being developed both at U.C. Berkeley and elsewhere, to experiment with or support alternative styles.

The Ptolemy kernel provides the most extensive support for domains where a design is represented as a network of blocks, as shown in figure 1-3. A base class in the kernel, called `Block`, represents an object in this network. Base classes are also provided for interconnecting blocks (`PortHole`) as well as for carrying data between blocks (`Geodesic`) and managing garbage collection efficiently (`Plasma`). Not all domains use these classes, but most current ones do, and hence can very effectively use this infrastructure.

Figure 1-3 shows some of the representative methods defined in these base classes. For example, note the *`initialize`*, *`run`*, and *`wrapup`* methods in the class `Block`. These provide an interface to whatever functionality the block provides, representing for example functions performed before, during, and after (respectively) the execution of the system.

Blocks can be hierarchical, as shown in figure 1-4. The lowest level of the hierarchy, as far as Ptolemy is concerned, is derived from a kernel base class called `Star`. A hierarchical block is a `Galaxy`, and a top-level system representation is a `Universe`.

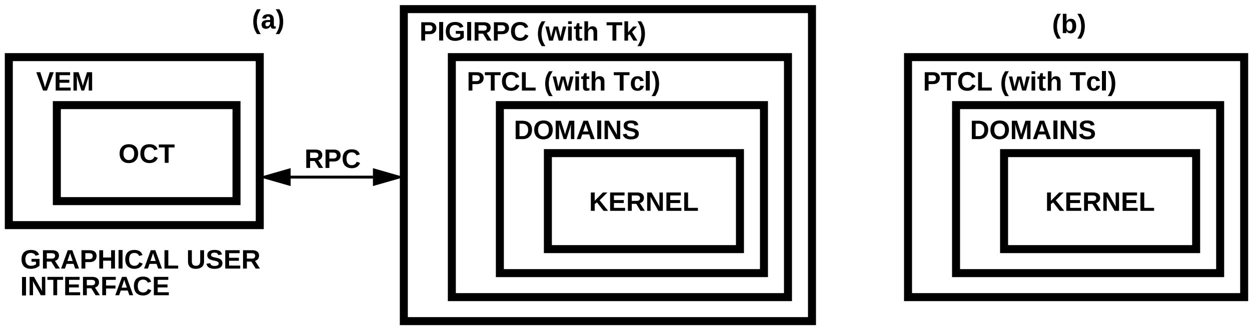


FIGURE 1-1: The overall organization of Ptolemy version 0.7, showing two possible execution styles: (a) graphical interface and (b) textual interface.

1.4 Models of Computation

The Ptolemy kernel does not define any model of computation. In particular, although the Berkeley team has done quite a bit of work with dataflow domains in Ptolemy, every effort has been made to keep dataflow semantics out of the kernel. Thus, for example, a network of blocks could just as easily represent a finite-state machine, where each block represents a state. It is up to a particular domain to define the semantics of a computational model.

Suppose we wish to define a new domain, called XXX. We would define a set of C++

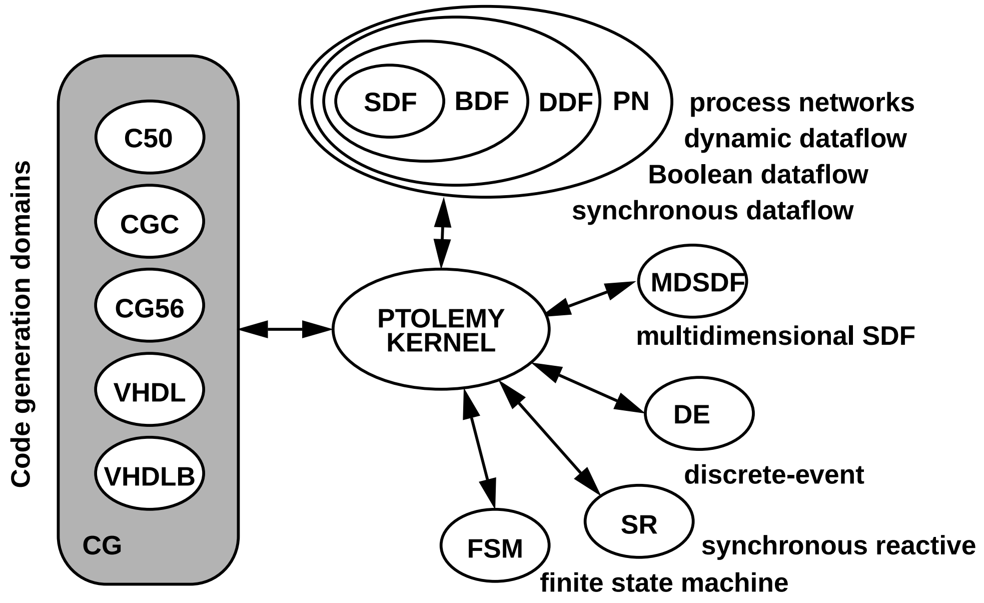


FIGURE 1-2: Domains available with Ptolemy 0.7

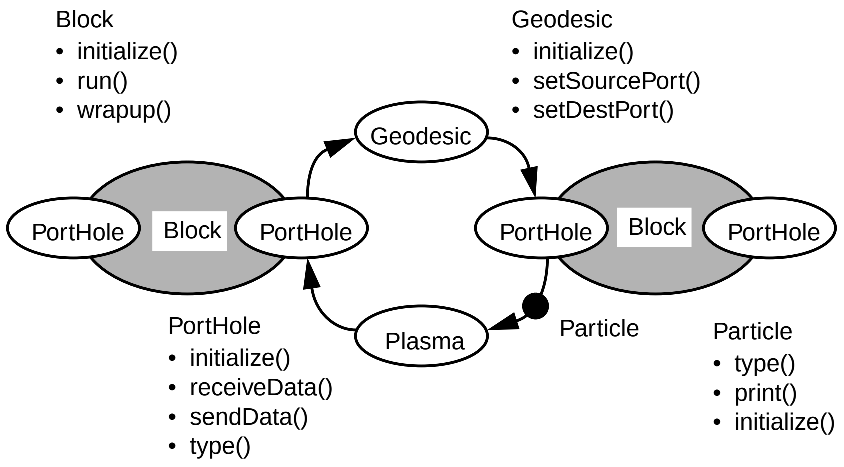
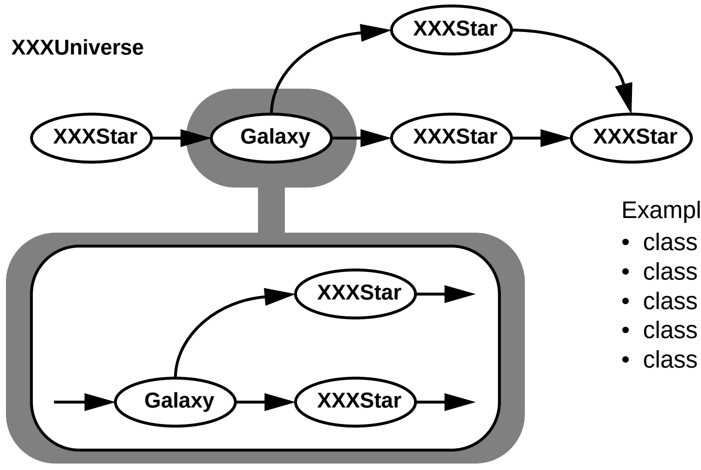


FIGURE 1-3: Block objects in Ptolemy can send and receive data encapsulated in Particles through Portholes. Buffering and transport is handled by the Geodesic and garbage collection by the Plasma. Some methods are shown.

classes derived from kernel base classes to support this domain. These classes might be called `XXXStar`, `XXXUniverse`, etc., as shown in figure 1-4.

The semantics of a domain are defined by classes that manage the execution of a specification. These classes could invoke a simulator, or could generate code, or could invoke a sophisticated compiler. The base class mechanisms to support this are shown in figure 1-5. A `Target` is the top-level manager of the execution. Similar to a `Block`, it has methods called `setup`, `run`, and `wrapup`. To define a simulation domain called `XXX`, for example, one would define at least one object derived from Target that runs the simulation. As suggested by figure 1-5, a Target can be quite sophisticated. It can, for example, partition a simulation for parallel execution, handing off the partitions to other Targets compatible with the domain.

A Target will typically perform its function via a Scheduler. The Scheduler defines the operational semantics of a domain by controlling the order of execution of functional modules. Sometimes, schedulers can be specialized. For instance, a subset of the dataflow model of computation called synchronous dataflow (SDF) allows all scheduling to be done at com-



Examples of Derived Classes
- class Star:: Block
- class XXXStar:: Star
- class Galaxy:: Block
- class Universe:: Galaxy, Runnable
- class XXXUniverse:: Universe

FIGURE 1-4: A complete Ptolemy application (a Universe) consists of a network of Blocks. Blocks may be Stars (atomic) or Galaxies (composite). The “XXX” prefix symbolizes a particular domain (or model of computation).

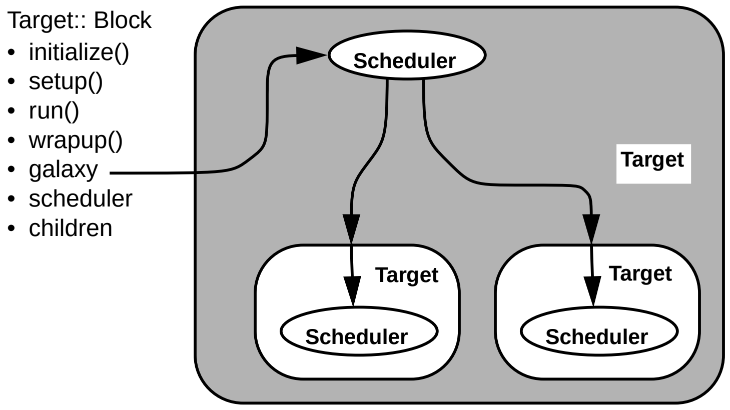


FIGURE 1-5: A Target, derived from Block, manages a simulation or synthesis execution. It ca invoke it’s own Scheduler on a Galaxy, which can in turn invoke Schedulers sub-Targets.

pile time. The Ptolemy kernel supports such specialization by allowing nested domains, as shown in figure 1-6. For example, the SDF domain (see figure 1-2) is a subdomain of the BDF domain. Thus, a scheduler in the BDF domain can handle all stars in the SDF domain, but a scheduler in the SDF domain may not be able to handle stars in the BDF domain. A domain may have more than one scheduler and more than one target.

1.5 Dataflow Models of Computation

One of the most mature domains included in the current system is the synchronous dataflow (SDF) domain [Lee87a,b], which is similar to that used in Gabriel. This domain is used for signal processing and communications algorithm development, and has particularly good support for multirate algorithms [Buc91]. It has been used at Berkeley for instruction, at both the graduate and undergraduate level [Lee92]. A dynamic dataflow (DDF) domain extends SDF by allowing data-dependent flow of control, as in Blosim. Boolean dataflow (BDF) [Buc93a,b,c] has a compile-time scheduler for dynamic dataflow graphs [Lee91a].

Several code-generation domains use dataflow semantics [Pin92][Mur93]. These domains are capable of synthesis of C code, assembly code for certain programmable DSPs [Won92], VHDL, and Silage [Kal93]. A significant part of the research that led to the development of these domains has been concerned with synthesizing code that is efficient enough for embedded systems [Bha93a,b,c][Bha94a,b][Buc93b,c]. A large amount of effort has also been put into the automatic parallelization of the code [Ha91][Ha92][Sih93a,b], and on parallel architectures that take advantage of it [Lee91b][Sri93].

A generalization of dataflow, called Kahn process networks [Kah74], has been realized by Tom Parks in the PN domain [Par95].

1.6 Discrete-Event Models of Computation

A number of simulation domains with discrete-event semantics has been developed for Ptolemy, but only the DE domain is released with Ptolemy 0.7. The DE domain is a generic discrete-event modeling environment, useful for simulating queueing systems, communication networks, and hardware systems. The discrete-event domains no longer released with

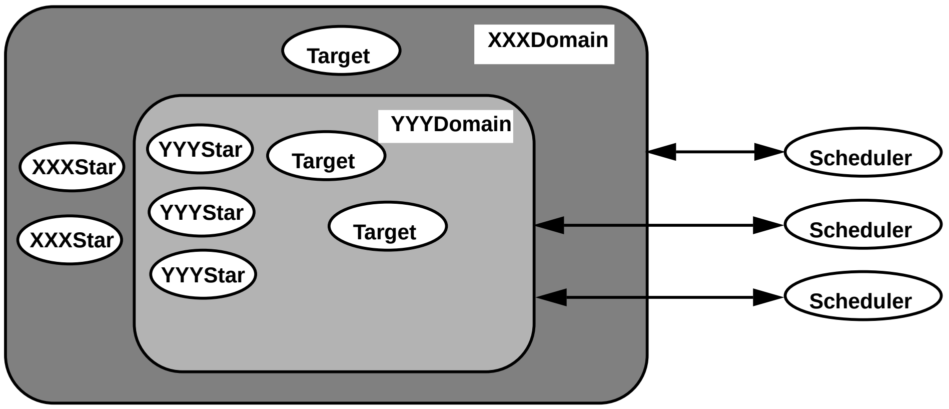


FIGURE 1-6: A Domain (XXX) consists of a set of Stars, Targets and Schedulers that support a particular model of computation. A sub-Domain (YYY) may support a more specialized model of computation.

Ptolemy 0.7 are Thor [Tho88] for modeling circuits at the register-transfer level [Kal93], communicating processes (CP) for modeling large-scale systems at a high level of abstraction, and message queue (MQ) for modeling a centralized network controller in a large-scale cell-relay network simulations [Lao94].

1.7 Synchronous Reactive Modeling

The software analogy of synchronous digital circuits has been realized by Stephen Edwards in the SR domain [Edw97]. This model of computation is better suited than dataflow to control-intensive applications, and is more efficient than DE.

1.8 Finite State Machines

Another approach to designing control-intensive applications is to mix the new FSM domain with dataflow, DE, or (in future releases) SR. The FSM domain is still very new and has many limitations, but we believe that for the long term, it provides one of the most exciting developments in the Ptolemy software.

1.9 Mixing Models of Computation

Large systems often mix hardware, software, and communication subsystems. The hardware subsystems may include pre-fabricated components, such as custom logic, processors with varying degrees of programmability, systolic arrays, and multiprocessor subsystems. Tools supporting each of these components are different, possibly using dataflow principles, regular iterative algorithms, communicating sequential processes, control/dataflow hybrids, functional languages, finite-state machines, and discrete-event system theory and simulation.

In Ptolemy, domains can be mixed and even nested. Thus, a system-level description can contain multiple subsystems that are designed or specified using different styles. The kernel support for this is shown in figure 1-7. An object called `XXXWormhole` in the `XXX` domain is derived from `XXXStar`, so that from the outside it looks just like a primitive in the `XXX`

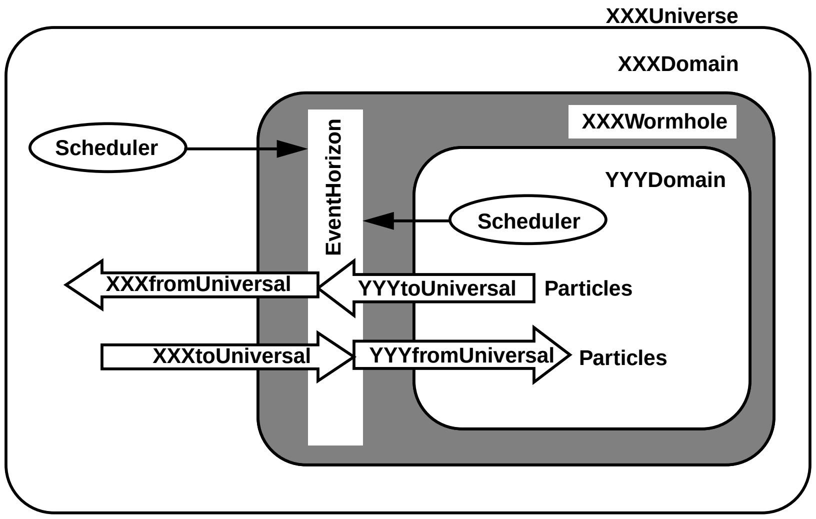


FIGURE 1-7: The universal EventHorizon provides an interface between the external and internal domains.

domain. Thus, the schedulers and targets of the `XXX` domain can handle it just as they would any other primitive block. However, inside, hidden from the `XXX` domain, is another complete subsystem defined in another domain, say `YYY`. That domain gets invoked through the `setup`, `run`, and `wrapup` methods of `XXXWormhole`. Thus, in a broad sense, the wormhole is polymorphic. The wormhole mechanism allows domains to be nested many levels deep, e.g. one could have a DE domain within an SDF domain within a BDF domain. The FSM domain is designed to always be used in combination with other domains.

1.10 Code Generation

Domains in figure 1-2 are divided into two classes: simulation and code generation. In simulation domains, a scheduler invokes the run methods of the blocks in a system specification, and those methods perform a function associated with the design. In code generation domains, the scheduler also invokes the run methods of the blocks, but these run methods synthesize code in some language. That is, they generate code to perform some function, rather than performing the function directly. The Target then is responsible for generating the connecting code between blocks (if any is needed). This mechanism is very simple, and language independent. We have released code generators for C, Motorola 56000 assembly, and VHDL languages, as show in figure 1-2.

An alternative mechanism that is supported but less exploited in current Ptolemy domains is for the target to analyze the network of blocks in a system specification and generate a single monolithic implementation. This is what we call compilation. In this case, the primitive blocks (`Stars`) must have functionality that is recognized by the target. In the previous code generation mechanisms, the functionality of the blocks is arbitrary and can be defined by the end user.

1.11 Conclusion

In summary, the key idea in the Ptolemy project is to mix models of computation, implementation languages, and design styles, rather than trying to develop one, all-encompassing technique. The rationale is that specialized design techniques are (1) more useful to the system-level designer, and (2) more amenable to high-quality high-level synthesis of hardware and software. The Ptolemy kernel demonstrates one way to mix tools that have fundamentally different semantics, and provides a laboratory for experimenting with such mixtures.

1.12 Current Directions

Since early 1995, a significant part of the Ptolemy project personnel have been pursuing models of computations for control-intensive computation, particularly in combination with compute-intensive subsystems, and mapping computation onto distributed architectures. The two primary models for control-intensive computation are finite state machines and a synchronous/reactive systems.

In late 1996, we shifted the focus of the project towards the design of distributed, network-aware, adaptive applications. We expect that future releases of our software will be network-savvy, including transparent HTTP support and mutable and migratable computations. Fundamental work in the semantics of models of computation will of course continue to fuel experiments with new domains and code generation techniques.

1.13 Organization of the documentation

The Ptolemy documentation is divided into three volumes. This volume, the first, is a user's manual. It is sufficient for users who do not plan to extend the system by adding code. It includes brief documentation of the most commonly used domains, and brief summaries of stars, galaxies, and demonstration programs that are distributed with the system.

The second volume is a programmer's manual. It includes chapters on writing new stars, writing targets, defining customized user interfaces by writing new Tcl/Tk code, and defining new domains. The third volume is the kernel manual. It details every C++ class defined in the Ptolemy kernel. It also gives full documentation for the classes supporting code generation. These classes provide the utilities used to build application-specific environments.

1.14 Acknowledgments

Ptolemy is a team effort in every sense. Here we acknowledge the key contributions, and apologize for inadvertent omissions.

1.14.1 Personnel

The overall coordinators are Prof. Edward A. Lee and Prof. David G. Messerschmitt of the EECS department at U. C. Berkeley, although there has also been involvement by the groups of Profs. Rabaey, Brodersen, Linnartz, Kahn, Sangiovanni, and Gray. Professional staff support has included Brian Evans, Alan Kamas, Christopher Hylands, John Reekie, Mary Stewart, Kirk Thege, and Kevin Zimmerman. Software organization and project management has been handled by Joseph Buck, Brian Evans, Alan Kamas, Christopher Hylands, Phil Lapsley, José Pino, John Reekie, and Kennard White.

Joseph Buck has been responsible for key management of the development of the kernel, and hence has impacted every aspect of Ptolemy. He also coordinated many of the contributions, and wrote the BDF domain, the interpreter (`ptcl`), and the original `ptlang` preprocessor. He also designed the memory allocation system used by assembly language code generation domains. Special thanks to Synopsys for allowing Joe to work on the 0.5 release after joining the company. Special thanks to Joe for his work on the 0.6 release.

Other key contributors to the kernel include Soonhoi Ha and Ichiro Kuroda. Soonhoi Ha also wrote the DDF, DE, and CGC domains, including many of the basic stars and the basic domain interface, and also made extensive contributions to the CG domain, the kernel, and parallel schedulers of all types. Anindo Banerjea and Ed Knightly wrote the DE Scheduler that is based on the calendar queue mechanism developed by Randy Brown. This was based on code written by Hui Zhang. Other significant contributions to the kernel have been made by Wan-Teh Chang, Mike Chen, Paul Haskell, Asawaree Kalavade, Alireza Khazeni, Tom Parks, José Pino, and Kennard White. Mike Chen wrote the matrix classes and the matrix particles, based in part on a prototype supplied by Chris Yu (from the Naval Research Laboratories). Mike Chen also developed the MDSDF domain. Joe Buck, Asawaree Kalavade, Alan Kamas, and Alireza Khazeni wrote the fixed-point particle class. Paul Haskell created the image particle classes and developed many of the image and video signal processing demos. Philip Bitar had impact on the design of the DE domain and on the visual style used in the graphical interface. Brian Evans developed the interfaces to MATLAB and Mathematica, with help from Steve Eddins at The MathWorks and Steve Gu, respectively.

All code generation domains are based on a secondary kernel implemented as the CG domain. Its principal creators are Joe Buck, Soonhoi Ha, Tom Parks, and José Pino. Kennard White made major extensions to the ptlang preprocessor to support code generation domains.

José Pino has been primarily responsible for assembly code generation domains, and Tom Parks for the C code generation domain, although extensive contributions have been made by Joe Buck, Soonhoi Ha, Christopher Hylands, Praveen Murthy, S. Sriram, and Kennard White. Chih-Tsung Huang, with help from José Pino, ported many of the assembly code generation stars from Gabriel. Many people had contributed to the Gabriel stars, including Jeff Bier, Martha Fratt, Wai Ho, Steve How, Phil Lapsley, Maureen O'Reilly, and Anthony Wong. Brian Evans and Luis Gutierrez have enhanced the Motorola 56000 stars, demonstrations, and targets, and S. Sriram has done the same for the Motorola 96000 stars. Patrick Warner wrote the C code generation target for the Network Of Workstations distributed operating system by Prof. Patterson's group at U.C. Berkeley.

Shuvra Bhattacharyya and Joe Buck wrote the loop scheduling mechanism, and Bhattacharyya contributed the Gantt chart display tool. The parallel schedulers were written by Gilbert Sih and Soonhoi Ha, with significant contributions from Joe Buck, Tom Parks, José Pino, and Kennard White. Praveen Murthy wrote the Sproc domain, used to generate parallel assembly code for the Sproc multiprocessor DSP, and Kennard White wrote the CM-5 target, used to generate parallel code for the connection machine from Thinking Machines, Inc.

Seungjun Lee and Tom Parks wrote the CP domain. Mike Williamson wrote the VHDL domains. Ichiro Kuroda from NEC contributed to the state handling mechanism.

The graphical user interface was written by Edwin Goei, based on the `vem` program, written by David Harrison and Rick Spickelmier. It has been extensively modified by Alan Kamas who has been responsible for the incorporation of Tcl/Tk into Ptolemy. The GUI has been enhanced by Wan-Teh Chang, Wei-Jen Huang, Mario Silva and Kennard White. Andrea Cassotto and Bill Bush have provided modifications and improvements to `vem`.

Christopher Hylands, Edward Lee, and John Reekie are the primary architects of the Tycho interface [Hyl97]. Tycho, named after the astronomer Tycho Brahe, is written in [Incr Tcl], an object-oriented extension of Tcl by Michael J. McLennan at AT&T Bell Labs. Significant development of Tycho has been contributed by Kevin Chang, Joel King, and Cliff Cordiero. Wan-Teh Chang and Bilung Lee have developed graphical editors for finite state machines. Code by Joseph Buck, Alan Kamas, and Douglas Niehaus originally written for pigi have been reused in the Tycho kernel. Some contributions to Tycho were made by Brian Evans.

Several people have had a major impact on the development of Ptolemy through their major efforts on its predecessor, Gabriel. Phil Lapsley has had incalculable impact on the directory structure, project management, documentation, and code generation efforts in Ptolemy. The first version of the graphical interface was written by Holly Heine.

Many people have had an impact on the current release by contributing stars and/or demo programs. These include, in addition to all the people mentioned above, Egbert Ammicht (from AT&T Bell Labs), Rachel Bowers, Stefan DeTroch (from IMEC), Rolando Diesta, Erick Hamilton, Wei-Yi Li, John Loh, and Gregory Walter. Others had an indirect impact by contributing stars or demo programs to the predecessor program, Gabriel. These include Jeff Bier, Martha Fratt, Eric Guntvedt, Mike Grimwood, Wai-Hung Ho, Steve How, Jonathan Lee, Brian Mountford, Maureen O'Reilly, Andria Wong, and Anthony Wong.

Ptolemy is very much an ongoing project, with current efforts expected to be included in future releases. Participants will be acknowledged when their work is included in a release.

1.14.2 Support

The Ptolemy project is currently supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Dolby Laboratories, Hewlett Packard, Hitachi, Hughes Space and Communications, LG Electronics, Lockheed Martin ATL, NEC, Philips, Rockwell, and the Semiconductor Research Corporation.

Funding at earlier stages of the project was also provided by the National Science Foundation (NSF), the Office of Naval Technology (ONT), via the Naval Research Labs (NRL), AT&T, Bell Northern Research (BNR), Hughes Network Systems, Hughes Research Laboratories, Mentor Graphics, Mitsubishi, Motorola, Sony, and Star Semiconductor.

In-kind contributions have been made by Ariel, Berkeley Camera Engineering, Philips, Spectrum Signal Processing, Synopsys, Signal Technology Inc. (STI), Texas Instruments, Wolfram Research, Inc., and Xilinx.

Other sponsors have contributed indirectly by supporting Gabriel, the predecessor.

1.14.3 Prior software

At every opportunity, we have built upon prior software, much of which we have been permitted to redistribute together with our Ptolemy distribution. We wish to gratefully acknowledge the following contributions:

- The `oct` tools, written by the CAD group at U.C. Berkeley, under the direction of Prof. Richard Newton, provide both the design database `oct` [Har86] and the graphical editor `vem` [Har86] for `oct`. The flexibility of `oct`, which makes minimal assumptions about the data stored in the database, and the extensibility of `vem`, through its `rpc` interface, have allowed us to use this software in ways unexpected by the authors.
- Tcl/Tk, architected by Prof. John Ousterhout of U.C. Berkeley, has improved the user interface in Ptolemy. The textual command-line Tcl interface [Ous90][Ous94] is the basis for the Ptolemy interpreter `ptcl`, and the graphics toolkit Tk [Ous94] is the basis for interactive graphics. Tcl serves a scripting language to control Ptolemy runs, and as an interpreter to compute parameters. Since Tcl is a scripting language, casual users have been able to extend Ptolemy's interface. Tcl is robust and lightweight.
- [Incr Tcl], an object-oriented extension of Tcl written by Michael J. McLennan at AT&T Bell Labs is used in Tycho.
- The Gnu tools, from the Free Software Foundation, have been instrumental in Ptolemy's development. The ability to distribute the compiler used in the development of Ptolemy has been critical to the success of our dynamic linking mechanism. It enables us to distribute a compiled executable together with the compiler that generated it. Thus, users lacking the skill or patience to recompile the Ptolemy system can nonetheless take advantage of dynamic linking of new functional blocks. They can use the same version of the compiler used to generate the executable, even if that version

of the compiler is not the one installed by default on their own system.

- Xgraph, written by David Harrison, of the CAD group at U. C. Berkeley, has provided the principal data display and presentation mechanism. Joe Buck modified this program only slightly, to accept binary input in addition to ASCII. Its flexibility and well conceived design have permitted us to use it for almost all data display. Only recently have we augmented it with Tk-based animated displays.

Chapter 2. The Interactive Graphical Interface

Authors: *Joseph T. Buck*
Edwin E. Goei
Wei-Jen Huang
Alan Kamas
Edward A. Lee

Other Contributors: *Andrea Cassotto*
Wan-Teh Chang
Michael J. Chen
Brian L. Evans
David Harrison
Holly Heine
Christopher Hylands
Tom Lane
Phil Lapsley
David G. Messerschmitt
Rick Spickelmier
Matthew Tavis

2.1 Introduction

The Ptolemy interactive graphical interface (`pigi`) is a design editor for Ptolemy applications. It is based on tools from the Berkeley CAD framework. In `pigi`, Ptolemy applications are constructed graphically, by connecting icons. Hierarchy is used to manage complexity, to abstract subsystem designs, and to mix domains (models of computation).

2.1.1 Setup

Ptolemy uses several environment variables (see page 2-51). In order for Ptolemy to run properly, the following two environment variables must be set in your `.cshrc` file:

- `PTOLEMY` is the full path name of the Ptolemy installation, and
- `PTARCH` is the type of computer on which you are running Ptolemy.

Example settings for a `.cshrc` file follow, along with how to update your path variable:

```
setenv PTOLEMY ~ptolemy
setenv PTARCH ‘$PTOLEMY/bin/ptarch‘
set path = ($PTOLEMY/bin $PTOLEMY/bin.$PTARCH $path)
```

When Ptolemy was installed, a fictitious user named ‘`ptolemy`’ may have been created whose home directory is the Ptolemy installation. If Ptolemy has been installed without creating a ‘`ptolemy`’ user, then use the appropriate path name of the Ptolemy installation for the value of the `PTOLEMY` environment variable, such as `/usr/eesww/share/ptolemy0.7`, for example. Once you make the appropriate changes to your `.cshrc` file, you will need to reevaluate the file:

```
source ~/.cshrc
```

In the documentation, we will generally refer to the home directory of the Ptolemy installation as `$PTOLEMY`, but sometimes we forget and use `~ptolemy`.

`Pigi` requires the MIT X Window System. If you are not familiar with this system, see the appendix, “Introduction to the X Window System” on page B-1. Some X window managers are configured to require that you click in a window before the “focus” moves to that window. This means that the window will not respond to input just because you have placed the mouse cursor inside it. You must first click a mouse button in the window. While it is possible to use `pigi` with this configuration, it is extremely unpleasant. In fact, it will be rather unpleasant to use *any* modern program that makes use of the window system. You will want to change the mode of the window manager so that the focus follows the mouse. The precise mechanism for doing this depends on the window manager. For the Motif window manager, `mwm`, the appropriate line in the `.Xdefaults` file is:

```
Mwm*keyboardFocusPolicy:   pointer
```

For the open-look window manager, `olwm`, the line is:

```
OpenWindows.SetInput:      followmouse
```

Alternatively, you can invoke `olwm` with the option `-follow`. Typically, the window manager is started in a file called `.xinitrc` in your home directory.

If you are running Sun’s OpenWindows, you may find that the Athena widgets have not been installed; `pigi` will not run without them. See the installation instructions in the appendix. For more information on using `pigi` with OpenWindows, see “Introduction to the X Window System” on page B-1.

2.2 Running the Ptolemy demos

A good way to start is by running a few of the Ptolemy demos. Any user can do this, although average users are not permitted to change the demos. If you feel compelled to change a demo, you can copy it to your own directory by using `cp -r` (see the section below, “Copying objects” on page 2-44). You can modify the copied version.

2.2.1 Starting Ptolemy

In any terminal window, change to the master demo directory:

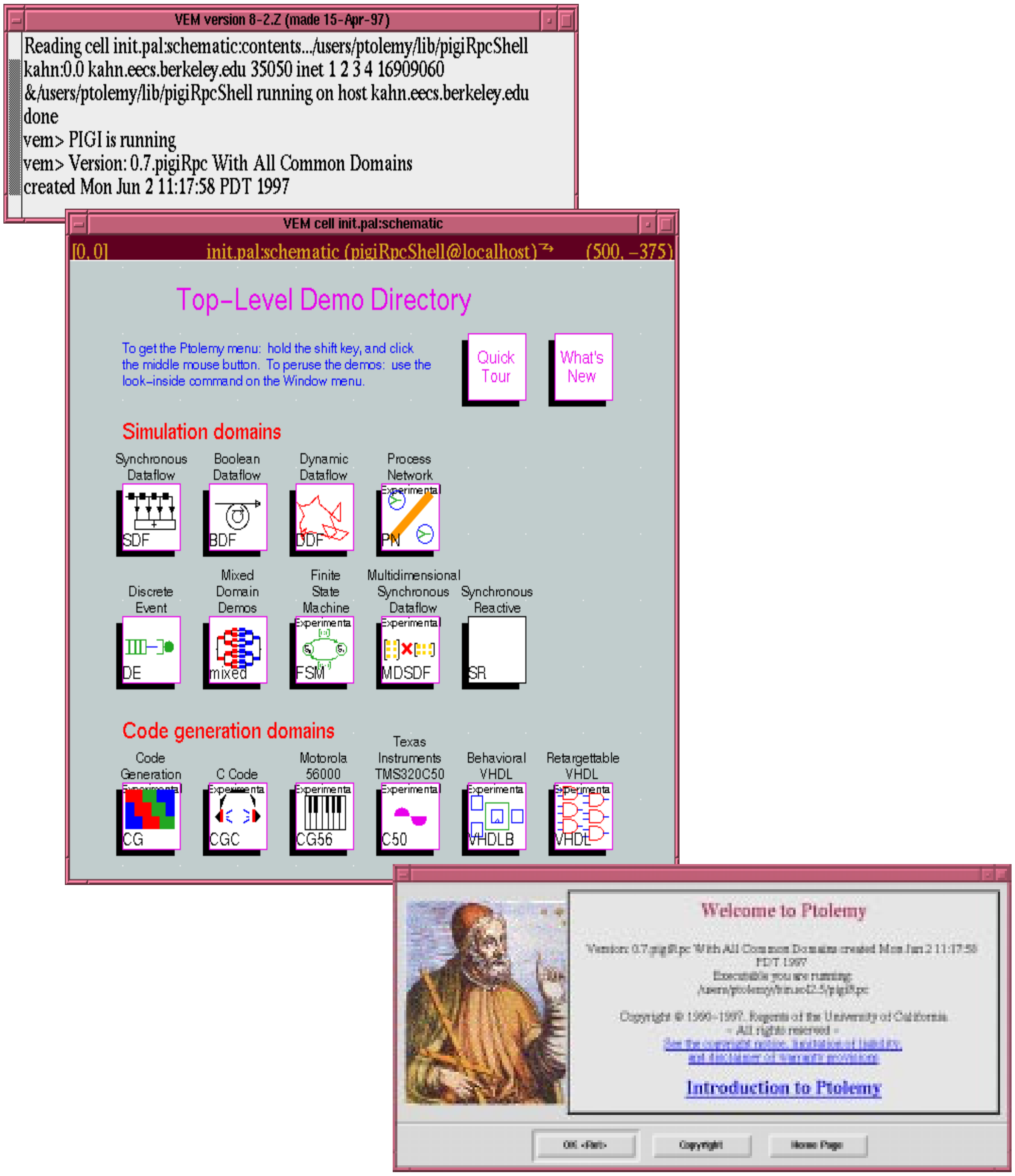
```
cd $PTOLEMY/demo
```

Start the Ptolemy graphical interface:

```
pigi &
```

You should get three windows: a `vem` console window at the upper left of your screen, a palette with icons of demonstrations below that, and a message window identifying the version of Ptolemy, as shown in figure 2-1. The borders on your windows may look different, since they are determined by the window manager that you use. If you have problems starting pigi, see

"Problems starting pigi" on page A-16. A complete list of options that you can specify on the command line is given in the section "Command-line options" on page 2-53. For example, if you are only interested in running the instructional/demonstration version, which only contains the Synchronous Dataflow and Discrete-Event Domains, then evaluate



VEM version 8-2.Z (made 15-Apr-97)

```
Reading cell init.pal:schematic:contents.../users/ptolemy/lib/pigiRpcShell
kahn:0.0 kahn.eecs.berkeley.edu 35050 inet 1 2 3 4 16909060
&/users/ptolemy/lib/pigiRpcShell running on host kahn.eecs.berkeley.edu
done
vem> PIGI is running
vem> Version: 0.7.pigiRpc With All Common Domains
created Mon Jun 2 11:17:58 PDT 1997
```

VEM cell init.pal:schematic

[0, 0] init.pal:schematic (pigiRpcShell@localhost) (500, -375)

Top-Level Demo Directory

To get the Ptolemy menu: hold the shift key, and click the middle mouse button. To peruse the demos: use the look-inside command on the Window menu.

Welcome to Ptolemy

Version: 0.7.pigiRpc With All Common Domains created Mon Jun 2 11:17:58 PDT 1997
Executable you are running:
/users/ptolemy/bin.sol2.5/pigiRpc

Copyright © 1990-1997, Regents of the University of California
- All rights reserved -
See the copyright notice, limitation of liability, and disclaimer of warranty provisions.

Introduction to Ptolemy

FIGURE 2-1: If you start pigi in the directory $PTOLEMY, once the system is started you will see these three windows. The upper left window is the vem console window. Below that is a palette of icons representing demo directories. To the right is the Ptolemy welcome window.

```
        pigi -ptiny &
```

Once you get all three windows, you have started two processes: the graphical editor, `vem`, and a process named `pigiRpc` that contains the `pigi` code and the Ptolemy kernel. The `vem` window prints the textual commands corresponding to your selections with the mouse. Watching the `vem` window is useful in diagnosing mistakes, such as drawing a box when you meant to draw a line. The `vem` console window also displays debugging messages, as well as the error and warning messages that appear in popup windows.

Clicking any mouse button in the welcome window (the one with the picture of Mr. Ptolemy) will dismiss it. Clicking the left mouse button on the "more information" button will display copyright information. The remaining windows can be moved and resized using whatever mechanism your window manager supports. The windows can be closed by typing a control-d with the mouse cursor inside the window. Closing the `vem` console window will terminate the entire program.

For reference, a summary of the pertinent terms is given in table 2-1 on page 2-4. These will be discussed in more detail as we go.

The palette window contains icons. Five different types of icons are used in `pigi`, as shown in figure 2-2. The ones in the palette window are of the first type; they represent other palettes. If you have a color monitor, the outline on these icons is purple.

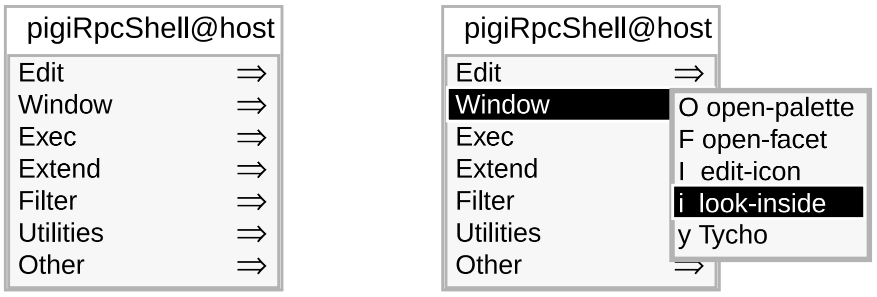
2.2.2 Exploring the menus

Place the mouse cursor on the icon labeled "SDF". Get the `pigi` command menu by holding the shift key and clicking the middle mouse button. This style of menu is called a "walking menu." Make sure you hold the shift button. The resulting command menu is shown

Category	Term	Definition
Programs	**Ptolemy**	The entire design environment
	pigi	The Ptolemy graphical interface, including both `pigiRpc` and `vem`
	`vem`	A graphical editor for oct, upon which `pigi` is built
	`pigiRpc`	A process (with remote procedure calls) attached to `vem` by `pigi`
Design database	`oct`	The design manager and database
	facet	A design object (a schematic or a palette)
	schematic	A block diagram
	palette	A facet that contains a library of icons rather than a schematic
Ptolemy objects	Star	Lowest level block in Ptolemy, with functionality defined in C++
	Galaxy	A block made up of connected sub-blocks, with inputs and/or outputs
	Universe	An outermost block representing a complete system that the user can run
	Domain	An object defining the model of computation, which defines the behavior of a network of blocks. In code generation, a domain also corresponds to single target language.
	Wormhole	A galaxy that does not have the same domain on the outside as the inside.
Tcl/Tk	Tcl	An interpreted language built in to `pigi`
	Tk	An X window toolkit attached to Tcl

TABLE 2-1: Summary of terms defining software components.

below:



The names displayed in the left main menu are only headers. To see the individual commands under each header, you must move the mouse to the arrows at the right of the menu. The sub-menu that appears on the right contains commands. Clicking any mouse button with a command highlighted as shown on the right will execute that command. To remove the menu without executing any command, simply click a mouse button anywhere outside the menu.

2.2.3 Traversing the hierarchy

Go to the "Window" sub-menu, and execute the *look-inside* command, as shown above on the right. A new palette will open, containing icons representing further palettes. Look inside the first of these, labeled "Basic". The icons inside contain application programs, called "universes" in Ptolemy. The two palettes you have just opened are shown in figure 2-3. They are both explained in further detail in "An overview of SDF demonstrations" on page 5-51.

Note in the Ptolemy menu that the *look-inside* directive has an "i" next to it. This is a "single key accelerator." Without using the walking menu, you can look inside any icon by simply placing the mouse cursor and hitting the "i" key on the keyboard. The single-key accelerators are extremely useful. In time, you will find that you use the menu only for commands that have no accelerator, or for which you cannot remember the accelerator. The Ptolemy commands obtained through the above menu are summarized in table 2-2. The few commands you will need immediately are shaded in table 2-2.

Look inside the first demo on the third row, labeled "`sinMod`". You will see the sche-

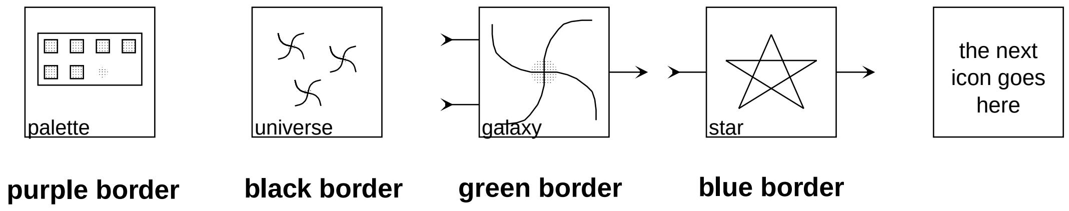


FIGURE 2-2: Five different types of icons are used in pigi. From left to right, the icons represent palettes (windows containing more icons), universes (windows containing Ptolemy applications), galaxies (functional blocks defined using other functional blocks), and stars (elementary or atomic functional blocks). The last icon on the right is the *cursor*, marking the position into which the next icon will be placed. On a color monitor, the borders of the icons have the indicated colors. The designs inside the icons and their shape are the default. They may be customized.

matic shown in figure 2-4. Try looking inside any of the icons in this schematic. If you look inside the icon labeled “modulator”, you will see the lower schematic in figure 2-4. If you look inside the icon labeled “XMgraph”, this time, instead of graphics, you will see text that defines the functionality of the block. The syntax of this text is explained in the programmer’s manual, volume 3 of the Almagest. You can change the editor used to display the text by setting an environment variable `PT_DISPLAY` (see “Environment variables” on page 2-51).

2.2.4 Running a Ptolemy application

To run the `sinMod` system using the walking menu, place the mouse cursor anywhere in the window containing the `sinMod` schematic, i.e., your cursor should be in the window

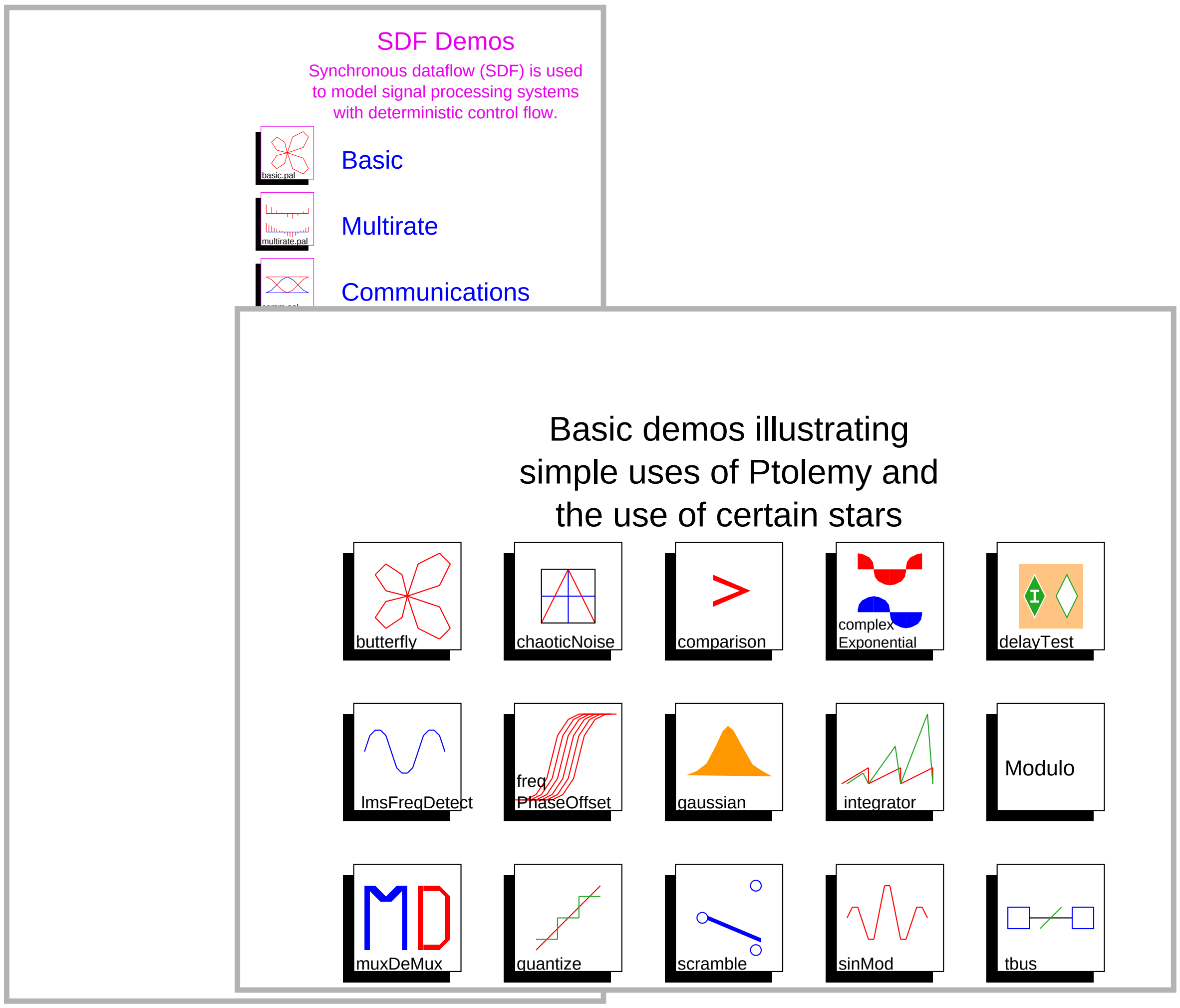
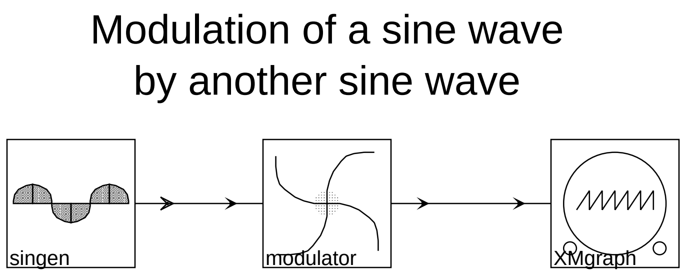


FIGURE 2-3: The “SDF” and “basic” palettes. The SDF palette contains icons representing other palettes containing a variety of demos in the synchronous dataflow domain. The “basic” palette is one such palette of demos. The icons here represent universes. These palettes are explained in more detail in “An overview of SDF demonstrations” on page 5-51

Menu	Heading	Command	Key	Description
pigi	Edit	edit-params	e	change parameters of a star, galaxy, or universe
		edit-domain	d	change the domain of a universe or galaxy
		edit-target	T	specify a target to manage the execution
		edit-comment	;	add comment to a universe or descriptor to a galaxy
		edit-pragmas	a	specify attributes of blocks
		edit-seed	#	set the random number seed
		find-name		highlight a block with a specified name
		clear-marks		clear all icon highlighting
	Window	open-palette	O	open one of the standard palettes of blocks
		open-facet	F	open an arbitrary palette, universe, or galaxy
		edit-icon	I	modify the physical appearance of an icon
		look-inside	i	look inside an icon for its definition
		Tycho	y	invoke the Tycho language-sensitive editor
	Exec	run	R	run a universe
		run-all-demos		testing command - run everything in a palette
		compile-facet		testing command - translate oct to Ptolemy
		display-schedule		show the most recent static schedule, if any
	Extend	make-schem-icon	@	make an icon to represent a facet
		make-star	*	dynamically link a new star and make an icon
		load-star	L	dynamically link a star that already has an icon
		load-star-perm	K	link a star so that derived stars can link dynamically
	Filter	equiripple FIR	<	invoke a provisional filter design utility
		window FIR	>	invoke another provisional filter design utility
	Utilities	plot signal	~	plot a signal read from a file
		plot Cx signal	-	plot a complex signal read from a file
		DFT	^	plot the DFT of a signal read from a file
		DFT of Cx signal	_	plot the DFT of a complex signal read from a file
	Other	facet number	H	testing command - display the Tcl facet handle
		man	M	open a manual page corresponding to a star
		profile	,	display a brief summary of the functionality of a star
		print-facet	cntr-P	print a facet or generate a PostScript file
		show-name	n	display the name of an icon and its master
		options		change various esoteric options
		version		display the version of Ptolemy that is running
		exit-pigi		quit Ptolemy without exiting vem

TABLE 2-2: A summary of the Ptolemy commands in the pigi menu, which is obtained by holding the shift button and clicking the middle mouse button. The single-key accelerators for commands that have them are shown. The commands that are most useful for exploring the Ptolemy demos are shaded.

that contains the following schematic:



Again holding the shift key, click the middle mouse button. Go to the “Exec” sub-menu, and select “run” by clicking any button. Notice that typing an “R” would have had the same effect.

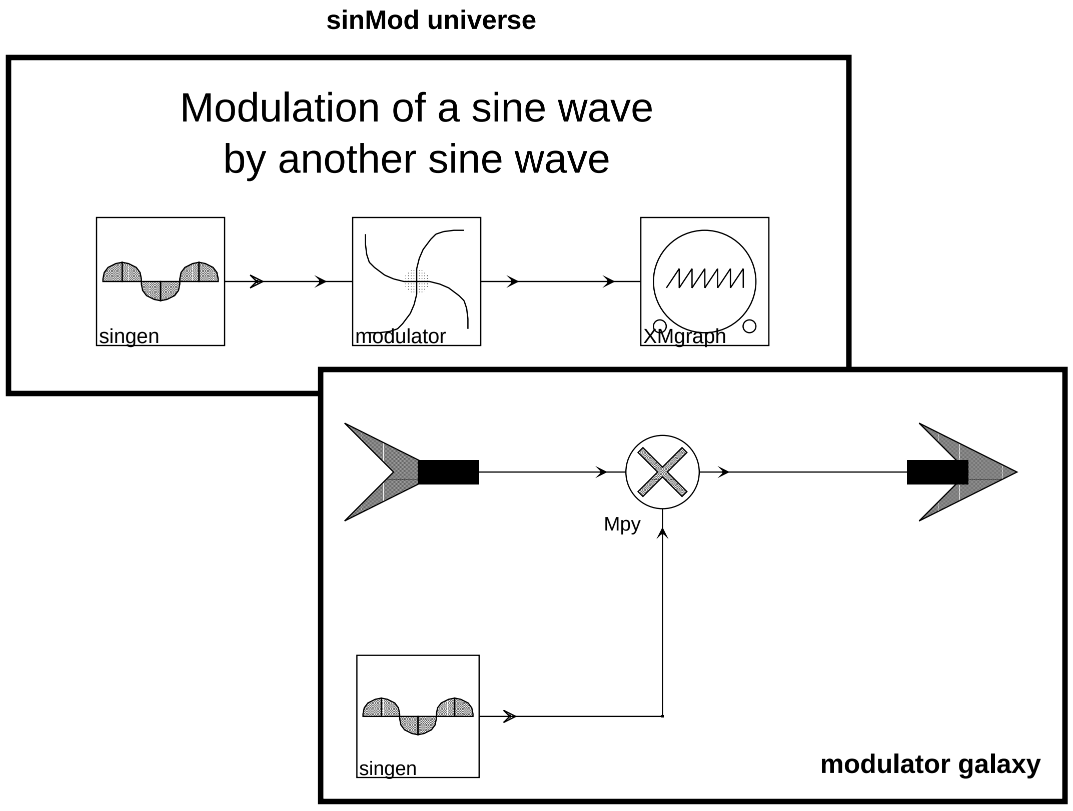
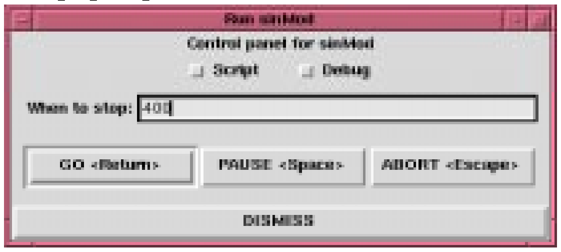


FIGURE 2-4: One of the synchronous dataflow demos. This Ptolemy application modulates a sine wave with another sine wave. The upper diagram is the top level. The lower is the contents of the “modulator” subsystem.

The following control panel pops up:



When to stop: 400

If you click the left mouse button on the “GO” button (or hit “return”), Ptolemy will run this application through 400 iterations. When the run is finished, a graph appears, as shown in figure 2-5. Try resizing and moving this display. Experiment in this `pxgraph` window by drawing boxes; to draw a box, just drag any mouse button. This causes a new window to open with a display of only the area that your box enclosed. Although the new window covers the old, if you move it out of the way, you can see both at once. Any of the now numerous open windows can be closed with a control-d.

2.2.5 Examining schematics more closely

Place the mouse cursor in any schematic or palette window, and click the middle mouse button without holding the shift key. The `vem` command menu, which is different from the `pigi` command menu, appears. This menu is the same style of “walking menu” as the

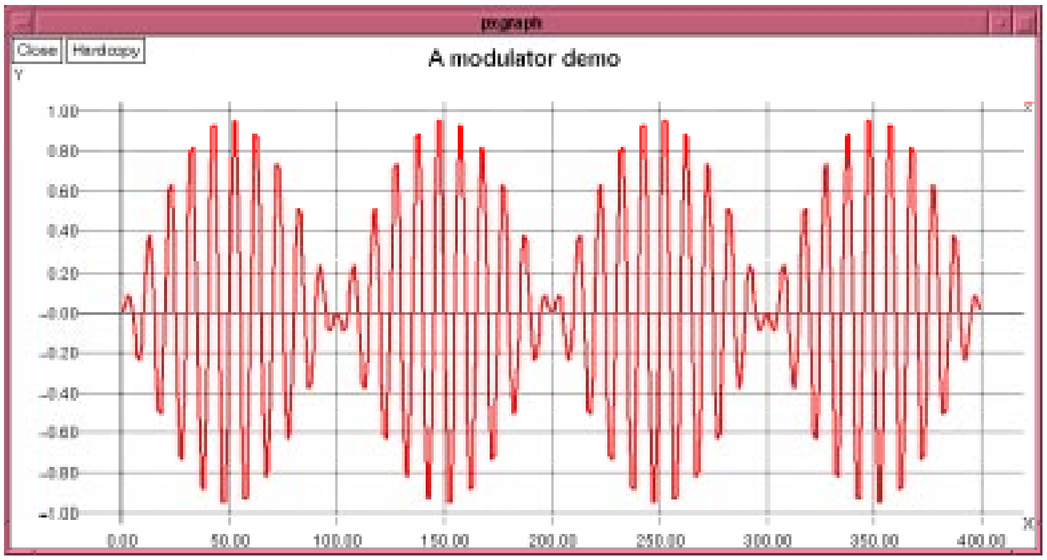
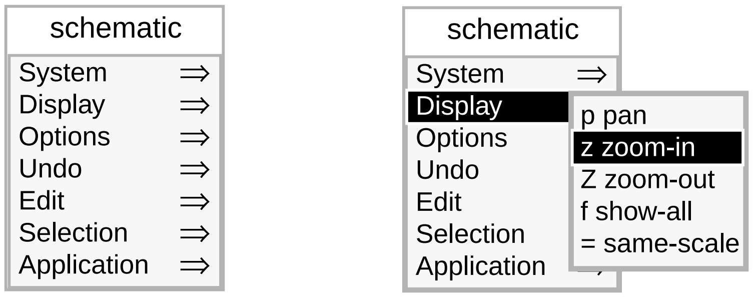


FIGURE 2-5: The graph generated by the “sinMod” application in figure 2-4. The graph is displayed by a program called “pxgraph,” based on xgraph by David Harrison.

`pigi` menu, and is shown below:



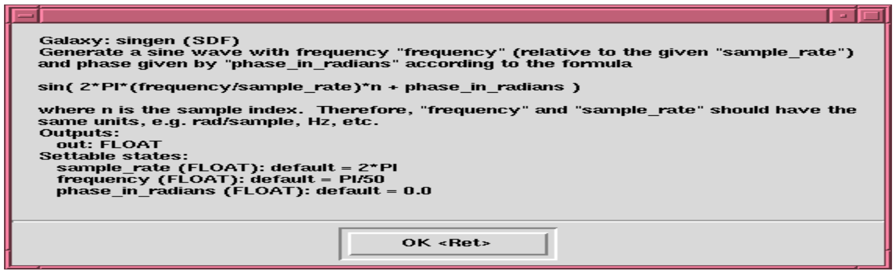
The `vem` menu is used for manipulating the graphical description of an application. The commands obtained through this menu are summarized in table 2-3, and explained in full detail in Chapter 19.

A few additional window manipulations will prove useful almost immediately. In any of the `vem` windows, you can closely examine any part of the window by drawing a box enclosing the area of interest and typing an “o”. Like in a `pxgraph` window, this causes a newwindow to open, showing only the enclosed area. Unlike `pxgraph` windows, typing the “o” is necessary. In addition, you can enlarge a window using your window manager manipulation, and type an “f” to fill the window with the schematic. You can also zoom-in (or magnify) by typing a “z”, and zoom-out by typing a “Z” (see table 2-3 on page 2-11). These and other `vem` commands are referenced again later, and documented completely in chapter 19.

2.2.6 Invoking on-line documentation for stars

You may wish to understand exactly how this `sinMod` example works. There are several clues to the functionality of the stars. After a while, the icons themselves will be all you will need. At this point, you can get several levels of detail about them. First, you will want to know the name of each star. If you have closed the `sinMod` window, open it again. Notice the names that appear on each of the icons. In more complicated schematics, when the icons are much smaller, the names will not show. You can zoom-in on a region of the window to see the names. Alternatively, you can place the mouse on any icon and issue the “show-name” command (in the “Other” menu), or type “n”.

Find the `singen` block at the left of the `sinMod` schematic. To understand its function, place the mouse cursor on it, and execute the Other:profile command. Here “Other” refers to the command category and “profile” to the command in the submenu (you may also type “,”). This command invokes a window that summarizes the behavior of the block, as shown below:



```
Galaxy: singen (SDF)
Generate a sine wave with frequency "frequency" (relative to the given "sample_rate")
and phase given by "phase_in_radians" according to the formula

sin( 2*PI*(frequency/sample_rate)*n + phase_in_radians )

where n is the sample index.  Therefore, "frequency" and "sample_rate" should have the
same units, e.g. rad/sample, Hz, etc.
Outputs:
  out: FLOAT
Settable states:
  sample_rate (FLOAT): default = 2*PI
  frequency (FLOAT): default = PI/50
  phase_in_radians (FLOAT): default = 0.0
```

For some blocks, further information can be obtained with the Other:man[1] (“M”) command, which displays a formatted manual page. Try it on the `XMgraph` block at the right of the schematic. The ultimate documentation for any block is, of course, its source code. For the `singen` block, the source is another schematic. Use the "look-inside" command (using the accelerator key "i") to see it. Recall that you can also look at the source code of the lowest level blocks (called *stars*) by looking inside them.

2.2.7 More extensive exploration of the demos

You can safely explore other demos in the palette by the same mechanism. The `but-`

Menu	Heading	Command	Key	Description
vem	none	no command name	cntr-h	remove the last argument (point, box, etc.)
			del	remove the last argument (point, box, etc.)
			cntr-u	remove all arguments from the argument list
			cntr-l	(control lower case L) redraw the window
	System	open-window	o	open a new view into a facet
		close-window	cntr-d	close a window
		where	?	find the position of the cursor in oct units
		palette	P	open the color palette for editing icons
		save-window	S	save a facet
		bindings	b	display key bindings (single key accelerators)
		re-read		restore a facet to the last saved version
	Display	pan	p	move the view to be centered at a given spot
		zoom-in	z	zoom in for a closer view of a facet
		zoom-out	Z	zoom out
		show-all	f	rescale the schematic to fit the window
		same-scale	=	used to get two windows to use the same scale
	Options	window-options		adjust snap, grid spacing, etc.
		layer-display		selectively display colors
		toggle-grid	g	turn on or off the grid display
	Undo	undo	U	undo any number of previous changes
	Edit	create	c	create a line, icon, name, etc.
		delete-objects	D	remove selected objects from an icon drawing
		edit-label	E	modify a label in a schematic
	Selection	select-objects	s	add an object to the argument list for a command
		select-net	cntr-N	select a wire (net) connecting blocks
		unselect-objects	u	remove an object from the argument list
		transform	t	rotate or reflect an object
		move-objects	m	move an object in a schematic
		copy-objects	x	copy one or more objects in a schematic
		delete-objects	D	delete objects from a schematic
	Application	rpc-any	r	start a vem application (pigiRpc is one)

TABLE 2-3: A summary of the Ptolemy commands in the vem menu, which is obtained by clicking the middle mouse button without holding the shift button. The single-key accelerators for commands that have them are shown. The commands that are most useful for exploring the Ptolemy demos are shaded. More complete documentation can be found in chapter 19, "Vem — The Graphical Editor for Oct" on page 19-1.

1. The man command uses Tycho to display the HTML format star documentation that is automatically generated by the `ptlang` program.

`terfly` demo at the upper left of the “basic” palette in figure 2-3 is particularly worthwhile. The demos in this and other palettes are briefly summarized in “An overview of SDF demonstrations” on page 5-51.

The `init.pal` palette in figure 2-1 contains icons leading to a top-level demo directory for each domain distributed with Ptolemy. Some of these are labeled “experimental”. These domains largely reflect research in progress and should be viewed as concept demonstrations only. The mature domains have no such label, although even these domains contain some experimental work. A quick tour of the basic capabilities can be had by looking inside the icon labeled “quick tour” in the start-up palette shown in figure 2-1. Each time you encounter a universe, run it.

2.2.8 What’s new

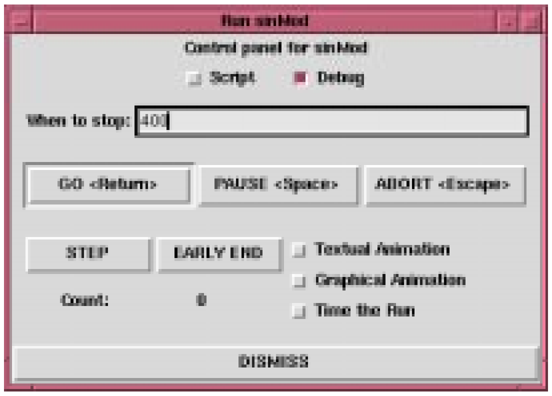
For readers familiar with previous versions of Ptolemy, you may wish to take a tour of the new features only. The “What’s New” icon in the `init.pal` palette in figure 2-1 leads to such a tour. Look inside it and you will see an icon for each of the last several releases. Open any one and explore the icons therein. Each time you encounter a universe, feel free to run it.

2.3 Dialog boxes

As you explore the demos, you will frequently encounter dialog boxes and control panels. For example, the run command opens a control panel like the one shown above that, among other things, allows you to specify how long the simulation should run. Most of the control panels that you will encounter have been designed using an X window toolkit called Tk, and every effort has been made to follow the Motif design style. Hopefully, this will look familiar to most people.

2.3.1 Tk control panels

Most of the items in a control panel are self-explanatory. Consider the run control panel shown on page 2-9. The button with the double relief (the GO button) is the default button. Hitting the return key has the same effect at clicking the mouse on this button. A different type of button is the “check button”, labeled “Debug”. Clicking on this button expands the control panel, as shown below, giving the user options that are sometimes useful in debugging a complex application.


Run sinMod
Control panel for sinMod
Script [✓ Debug]
When to stop: 400
Count: 0

The “Animation” buttons show (textually or graphically) which blocks are running at any given time. Graphical animation will dramatically slow down a simulation, so it is not advised except for occasional use. It is often useful in combination with the `STEP` button, which will fire stars one a time.

The `EARLY END` button terminates the simulation as of the point currently reached, but then it runs the `wrapup` methods of the stars, just as if the simulation had ended normally. Thus, it is an invasive alteration of the behavior of the simulation. The results displayed during `wrapup` may be subtly or wildly different from the results that would have been obtained if the simulation had been allowed to proceed to its scheduled end time. Some of the demos will in fact deliver incorrect, or at least unexpected, results if stopped early.

The `EARLY END` button differs from the `ABORT` button in that the `EARLY END` button calls the `wrapup` methods, `ABORT` does not. Thus, for example, signal plots that normally appear at the end of a simulation will not appear when `ABORT` is used.

Clicking on the `Debug` button a second time will reduce the control panel to its previous form.

Many control panels have text widgets. In the control panel above, for example, the box labeled “When to stop” is a text widget. To change the number, you must use Emacs-like editing control characters. These are summarized in table 2-4. In addition, using the mouse, you can position the cursor anywhere in the text to begin editing by clicking the left button. For example, to enter a new number for “when to stop”, position the cursor in the number box and type control-k followed by the new number. You can then push the GO button (or type return) to run the application the specified number of iterations.

Many control panels have more than one text widget. The current field is the one with the cursor, and anything you type will go into it. To change the current field to a different one, move the mouse or use the “Tab” key to move to the next one.

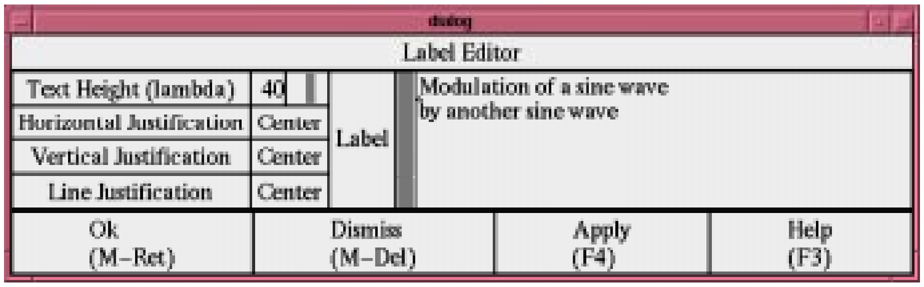
2.3.2 Athena widget dialog boxes

Although we have been working hard to eliminate them, a few old-style dialog boxes based on the Athena widgets from MIT still survive in the system. You will recognize these immediately because they are much uglier and more difficult to work with than the Tk-based

Key	Description
Delete, control-h	Delete previous character.
control-a	Move to beginning of line
control-b	Move backward one character
control-d	Delete next character
control-e	Move to end of line
control-f	Move forward one character
control-k	Kill (delete) to end of line

TABLE 2-4: Summary of key bindings for Emacs-style text editing.

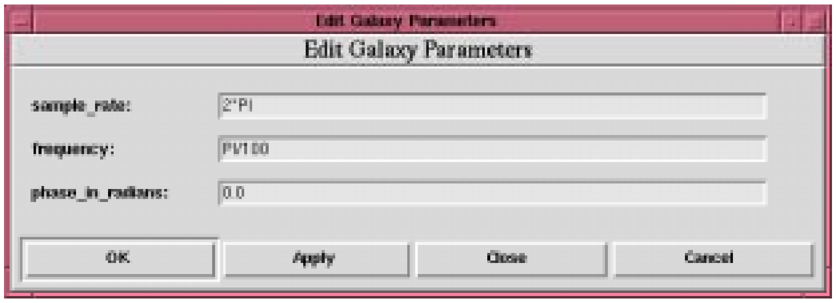
widgets. Here is an example:



The text widgets in these dialog boxes also use Emacs-style commands. However, do not type return; this adds a second line to the dialog entry, which for most commands is confusing at best. If you accidentally type return, you can backspace sufficiently to get back to one line. Meta-return is the standard way to invoke the "OK" button in these widgets.

2.4 Parameters and states

To see the parameter values of a star or galaxy, execute the Edit:edit-params command, which has the accelerator key "e". The `singen` star in the `sinMod` application has the following parameter screen:



Notice that the *frequency* parameter is given as an expression, "PI/100" (PI represents the constant π). This section describes the expression language for specifying parameter values.

The parameter screen can be kept open while you experiment with different values of the parameters. Try changing the value "PI/100" to "PI/200". Click "Apply" in the parameter window, and then "GO" in the run control panel. How does this change the display? Clicking "Cancel" in the parameter window will restore the parameter values to the last saved values and dismiss the parameter window. Clicking "Close" will dismiss the parameter window without restoring the parameter values.

2.4.1 A note on terminology

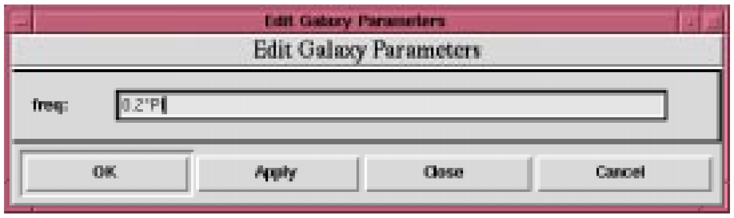
A `State` is a data-structure associated with a star and is used to remember data values from one invocation to the next. For example, the gain of an automatic gain control is a state. A state need not be dynamic since its value may not change during the course of a simulation. Technically, a *parameter* is the initial value of a state. `Pigi` is responsible for defining parameter values and storing them in the design database.

2.4.2 Changing or setting parameters

The *edit-params* command in `pigi` permits the user to set the initial value of a settable state of any star (lowest level block) and to define and set parameters for a galaxy (composite block) or universe (complete application).

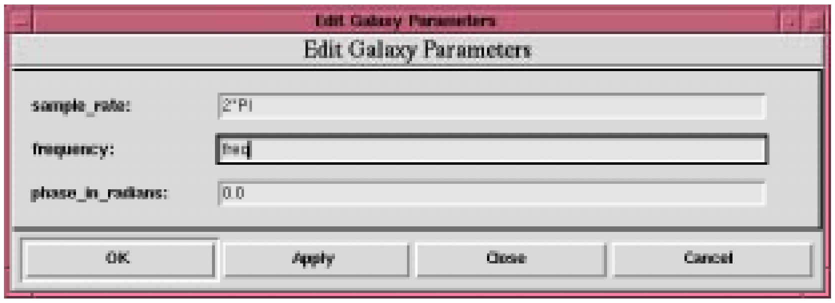
Passing parameters through the hierarchy

Star parameters may be linked to the parameters of the galaxy or universe that contains the star. The syntax for linking the values of the star parameters to values of galaxy or universe parameters is simple. Consider again the `sinMod` application shown in figure 2-4. The parameter screen for the `modulator` block is shown below:



freq: 0.2*PI

This block, however, is a galaxy, not a star. If you look inside (as has been done in figure 2-4), and edit the parameters of the `singen` block inside `modulator`, you will see



sample_rate: 2*PI
frequency: freq
phase_in_radians: 0.0

Notice now that the value of the *frequency* parameter is a symbolic expression, "freq". This refers to the galaxy parameter "freq". Thus, parameter values can be passed down through the hierarchy. These symbolic references can appear in expressions, which we discuss next.

Parameter expressions

Parameter values set through `pigi` can be arithmetic expressions. This is particularly useful for propagating values down from a universe parameter to star parameters somewhere down in the hierarchy. An example of a valid parameter expression is:

```
PI/(2*order)
```

where `order` is a parameter defined in the galaxy or universe. The basic arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). These operators work on integers and floating-point numbers. Currently all intermediate expressions are converted to the type of the parameter being computed. Hence, it is necessary to be very careful when, for example, using floating-point values to compute an integer parameter. In an integer parameter specification, all intermediate expressions will be converted to integers.

Complex-valued parameters

When defining complex values, the basic syntax is

```
(real, imag)
```

where `real` and `imag` evaluate to integers or floats.

Fixed-point parameters

Fixed-point parameters may be assigned a precision directly. To do this, the parameter is given in the syntax “(*value*, *precision*)”, where *value* is an ordinary number and *precision* is given by either of two syntaxes:

- **Syntax 1**: As a string like “3.2”, or more generally “*m.n*”, where *m* is the number of integer bits (to the left of the binary point) and *n* is the number of fractional bits (to the right of the binary point). Thus length is *m*+*n*.
- **Syntax 2**: A string like “24/32” which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one.

Thus, for example, a fixed-point parameter might be defined as “(0.8, 2/4).” This means that a 4-bit word will be used with two fraction bits. Since the value “0.8” cannot be represented precisely in this precision, the actual value of the parameter will be rounded to “0.75”.

A fixed-point parameter can also be given a value without a precision. In this case, the default precision is used. This has a total word length of 24 bits with the number of integer bits set as required to store the value. For example, the number 1.0 creates a fixed-point object with precision 2.22, and a value like 0.5 would create one with precision 1.23.

The precision of internal computations in a star is typically given by a parameter of type `precision`. A precision parameter has a value specified using either of the two syntaxes above.

2.4.3 Reading Parameter Values From Files

The values of most parameter types can be read from a file. This syntax for this is to use the symbol < as in the following example:

```
< filename
```

First, any parameters appearing in the `filename` in the form of `{parameter}` are replaced with their values. Then, any references to environment variables or home directories are substituted to generate a complete path name. Finally, the contents of the file are then read and spliced into the parameter expression and reparsed. File inputs can be very useful for array parameters which may require a large amount of data. Other expression may come before or after the `<filename` syntax (any white space that appears after the < character is ignored).

2.4.4 Inserting Comments in Parameters

Comments are also supported for non-string parameters. A comment is specified with the # symbol. Everything after the # until the end of the line is discarded when the parameter is evaluated. Comments are especially useful in combination with files as they can help remind the user of which galaxy or star parameter the file was written.

For example, a comment could be added to the *frequency* parameter above:

```
freq # This is set to the Galaxy parameter
```

Comments are not supported for the String parameter or String Array parameter types. In fact, when the image processing stars use String states to represent a filename, the # character is used to denote the frame number of the image being processed.

2.4.5 Using Tcl Expressions in Parameters

Arbitrary Tcl expressions can be embedded in a parameter expression by preceding the expression with the `!` character as in the following example:

```
! "expression"
```

First, parameters in the form of `{parameter}` appearing in the expression are replaced by their values. Then, the string is sent to the pigiRpc Tcl interpreter for evaluation. Finally, the result is spliced into the parameter expression and reparsed. The pigiRpc Tcl interpreter is the same interpreter that appears as a window when `pigi` is started by using `pigi -console`.

This facility is general and supports both numeric and symbolic computing of expressions. Through Tcl, one can access all of its math functions, which generally behave as the ANSI C functions of the same name: `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `double`, `exp`, `floor`, `fmod`, `hypot`, `int`, `log`, `log10`, `pow`, `round`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`. So, a parameter expression could be

```
! "expr sqrt(2.0 / {BitDuration})"
```

for the amplitude of the oscillators in a binary frequency shift keying system, in which `BitDuration` is a parameter. The `expr` command is a Tcl command that treats its arguments as a single mathematical expression that must evaluate to a number.

The Tcl mechanism can be used to return symbolic expressions:

```
! "join 2*gain1"
```

Because `gain1` is not surrounded by curly braces, its value is not substituted before passing the expression to the Tcl interpreter. The Tcl interpreter will return `2*gain1` which is then evaluated by the parameter parser.

Note that whitespace between `!` and `"` is permitted in numeric parameters, but not in string parameters: to get a Tcl call to be recognized in a string parameter you must write:

```
!"list /users/ptolemy/myfile"
```

There are several Tcl commands embedded in `pigiRpc` that help support parameter calculations. They are: `listApplyExpression`, `max`, `min`, `range`, `rangeApplyExpression`, and `sign`. For example,

```
! "min [max 1 2 3] [sign -2]"
```

first evaluates to `min 3 -1` and then to `-1`. The procedure `range` returns a consecutive sequence of numbers:

```
! "range 0 5"
```

returns `0 1 2 3 4 5`. The `rangeApplyExpression` procedure generates a sequence of values by applying a consecutive sequence of numbers to a Tcl expression that is a function of `i`. For example, you can generate the taps of an FIR filter that is a sampled sinusoid by using

```
! "rangeApplyExpression { cos(2*{PI}*$i/5) } 0 4"
```

generates one period of sinusoidal function and returns

```
1.0 0.309042 -0.808986 -0.809064 0.308916
```

The `listApplyExpression` is similar to `rangeApplyExpression` except that it only takes two arguments: the second argument is a list of numbers to substitute for `i` in the expression. The command

```
! "listApplyExpression { cos(2*{PI}*$i/5) } [range 0 4]"
```

is equivalent to the previous example of the `rangeApplyExpression` function.

If you are running Tycho TclShell from within `pigi` or `pigi -console`, you can receive help on the new Tcl procedures `listApplyExpression`, `max`, `min`, `range`, `rangeApplyExpression`, and `sign`, by typing

```
help sign
```

at the prompt. To start Tycho from within `pigi`, type a `y` while the mouse is over a `vem` facet or palette.

The Tycho TclShell and the `pigiRpc` console includes the Ptolemy interpreter (`ptcl`) which defines the help mechanism. Help is available on all of the commands we have added to the Tcl language.

2.4.6 Using Matlab and Mathematica to Compute Parameters

Since Tcl can be used to compute parameters as described in the previous section, Ptolemy's Tcl interface to Matlab [Han96] and Mathematica [Wol91][Bla92] can be used to compute parameters. This allows even more expressiveness, but the drawback is that demonstrations relying on Matlab and Mathematica will only work at sites that have Matlab and Mathematica installed. For example, we can use Matlab to design an 32-order FIR half-band filter using the Parks-McClellan optimal equiripple FIR filter design algorithm:

```
! "matlab getpairs c {c=remez(32, [0 0.4 0.6 1], [1 1 0 0])}"
```

Similarly, we can use Mathematica to derive formulas to be used as parameters:

```
! "mathematica get c {c=Integrate[A x, {x, 0, 1}]}"
```

This command returns the symbolic expression `A/2` which is reparsed by Ptolemy. Matlab and Mathematica can be used to keep track of how parameter values are computed. Mathematica can also be used to return symbolic expressions that can be used in conjunction with higher-order functions to define scalable systems [Eva95].

The Ptolemy interface to Matlab and Mathematica can also be accessed from the pigiRpc console window, and the Tycho editor offers console windows that mimic the Matlab and Mathematica teletype (tty) interfaces. More information about the options of the Tcl commands `matlab` and `mathematica` can be found by using the help facility described above.

2.4.7 Array parameters

When defining arrays of integers, floats, complex numbers, fixed-point numbers, or strings, the basic syntax is a simple list separated by spaces. For example,

```
1 2 3 4 5
```

defines an integer array with five elements. The elements can be expressions if they are surrounded by parentheses:

```
1 2 PI (2*PI)
```

Repetition can be indicated using the following syntax:

```
value[n]
```

where `n` evaluates to an integer. An array or portion of an array can be input from a file using the symbol < as in the following example:

```
1 2 < filename 3 4
```

Here the first two elements of the array will be 1 and 2, the next elements will be read from file `filename`, and the last two elements will be 3 and 4. This latter capability can be used in combination with the `WaveForm` star to read a signal from a file.

2.4.8 String Parameters

There is a bit of complication when one wishes to set a string parameter or string array parameter equal to the value of a galaxy or universe parameter. This is because a distinction must be made between a sequence of characters that give the name of a symbol and a sequence of characters to be interpreted literally. The syntax to use is explained in the example:

```
This string has the word {word} taken from another parameter
```

Here `{word}` represents the value of a string universe or galaxy parameter. This capability is especially useful for constructing labels for output plots. When using string states to specify options for a Unix command, as in the options parameter in `Xgraph` stars, you can use either double quotes or single quotes to include white space within a single word:

```
-0 'original signal' -1 'estimated signal'
```

String arrays have a few more special restrictions. Each word (separated by white space) is a separate entry in the array. To include white space in an element of the array, use quotation marks. Thus, the following string array

```
first "the second element" third
```

has three elements in it. The string array

```
repeat[10]
```

has ten separate copies of the string “repeat” in 10 separate entries in the array. Curly braces are used to substitute in values from galaxy parameters. Thus, in

```
{paramname}
```

`paramname` must be the name of either a string array or a scalar-valued parameter (an integer, float or complex array, for example, is not permitted). If it is a string array, then each element of `paramname` becomes an element of the parameter. If it is some other kind of parameter the value becomes a single element of the string array.

To use one of `[`, `]`, `{`, or `}` literally, quote them with double quotes. To turn off the special meaning of a double quote, precede it with a backslash: `\"`. Similarly, use `\\` to get a single backslash.

String array values may also be read from files using the < symbol. For details on how to use file references, see section 2.4.3 above. Note that for string arrays, the filename can be a literal string such as

```
< $PTOLEMY/data/filename
```

as well as a string that refers to parameters such as

```
< $PTOLEMY/{data_dir}/data_file
```

in which case the value of the parameter *data_dir* would be substituted. Ptolemy does not perform expansion of filenames such as `file.{1,2}` into `file1 file2` as a Unix shell might do.

2.5 Particle types

The packets of data that pass from one star to another in Ptolemy are called *particles*. So far, all particles have simply been floating-point numbers representing samples of signals. However, several other data types are supported. Each star icon has a stem for each porthole. In `pigi`, if you are using a color monitor, the color of the stem indicates the type of data that the porthole consumes or produces, as summarized in table 2-5. A blue stem on an input or output of a star icon indicates type “float”, a purple stem indicates type “fix” for fix-point particles, a white stem indicates type “complex”, an orange stem indicates type “int” for integer particles, a green stem indicates “message”, a black stem indicates type “string”, a yellow stem indicates type “file”, and a red stem indicates “anytype”. The “message” type is a user-defined data type (see the programmer’s manual). A star that operates on “anytype” particles is said to be *polymorphic*. Polymorphic stars operate on multiple types of data. For example, a `Printer` star can produce a textual representation of any type of particle. In addition, stars

Type name	Stem Color	Description
ANYTYPE	red	any data type is accepted
FLOAT	blue	floating-point scalars
FLOAT_MATRIX_ENV	blue (thick)	floating-point matrices
COMPLEX	white	complex scalars
COMPLEX_MATRIX_ENV	white (thick)	complex matrix
INT	orange	integer scalar
INT_MATRIX_ENV	orange (thick)	integer matrix
FIX	violet	fixed-point scalar
FIX_MATRIX_ENV	violet (thick)	fixed-point matrices
MESSAGE	green	user-defined data type
STRING	black	string
FILE	yellow	filename

TABLE 2-5: Data types supported by the Ptolemy kernel.

which input or output Matrix type particles have stems which are extra thick with colors corresponding to the four main types, float, int, complex, and fix.

Ptolemy usually makes conversions between numeric particle types automatically. The float to complex conversion does the obvious thing, putting the float value into the real part of the complex number and setting the imaginary part to zero. The complex to float conversion computes the magnitude of the complex number. Int to float is easy enough. Float to int rounds to the nearest integer.

The `Xscope` star, and some other stars that generate output, accept “anytype” of input. However `Xscope` isn’t completely polymorphic, because it converts all inputs to float internally. So for a complex input, the magnitude will be plotted. If you want to plot both the real and imaginary parts you should use the `ComplexReal` conversion star first.

In some situations automatic type conversions cannot be made. A common difficulty involves several outputs of different types feeding a `Merge` star. Ptolemy must assign a specific type to the `Merge` star’s output, but in this case it will be unable to decide which type to use, so it will complain that it “can’t determine DataType” for the output. The solution is to insert one or more type conversion stars, so that all the values arriving at the `Merge` star have the same type. (The type conversion stars can be found in the “conversion” palette of the appropriate domain. It will be explained below how to find this.)

There are no automatic conversions between matrix particles and scalar particles; in fact the matrix particle types do not support automatic type conversion at all. Conversion stars need to be explicitly inserted between two stars that work on different Matrix types.

Some domains are more restrictive about particle type conversions than others. Assignment of types to ANYTYPE portholes and resolution of type conflicts is discussed further in section 4.6 of the Ptolemy Programmer’s Manual, and in the Ptolemy Kernel Manual.

2.6 The oct design database and its editor, vem

With the experience gained so far, it may be helpful to explain more clearly the software architecture of the system. `Pigi` is built on top of existing CAD tools that are part of the Berkeley CAD framework. An important component of this framework is `oct`, which serves as the design database. `Oct` keeps track of block connections, parameter values, hierarchy, and file structure, and hence moderates all accesses to designs stored on disk. The organization is shown in figure 2-6. `Vem` is an interactive graphical editor for `oct`. `Vem` provides one of many ways to examine and edit designs stored by `oct`. This chapter gives just enough information about `vem` to use it with Ptolemy in simple ways. More complete documentation is contained in chapter 19, “Vem — The Graphical Editor for Oct” on page 19-1.

In `pigi`, the Ptolemy kernel runs in a separate Unix process, called `pigiRpc`, attached to `vem`. Users edit designs using `vem`, store their designs using `oct`, and execute their application through the link to the Ptolemy kernel. The two Unix processes are shown in the shaded boxes in figure 2-6. The user interacts with both processes but only the user interface of the `pigiRpc` process has been upgraded to use Tcl/Tk, as explained above. With this software architecture in mind, we can now define terms that we have been using informally.

`Oct` objects (which are stored on disk) are called facets. A *facet* is the fundamental unit that a user edits with `vem`. As an analogy, we can think of a facet as a text file in a computer system and `vem` as a text editor, such as `vi` or `emacs`. However, instead of calling system routines to access the data stored in a text file like `vi` does, `vem` calls `oct` routines to access the data stored in a facet. Thus, `oct` manages all data accesses to facets. Facets may define a universe or a galaxy, for example. Thus, figure 2-4 on page 2-8 shows a facet that defines a universe and a second one that defines a galaxy.

Facets may also define the physical appearance and formal terminals of icons that represent stars, galaxies, universes, and wormholes, e.g., the physical appearance of each icon in figure 2-4 is defined in another facet called the *interface facet*. A schematic that uses icons, by contrast, is called a *contents facet.* The "edit-icon" command ("I") will open the facet defining an icon. Instructions for modifying the appearance of an icon are given in "Editing Icons" on page 2-34.

A facet may also contain a *palette*, which is simply a collection of disconnected icons. Palettes are directories of stars, galaxies, and universes in a library. Thus, for example, figure 2-3 on page 2-6 shows two palettes, both of which contain sets of icons. Note that facet names, like file names in Unix, should not contain spaces.

2.7 Creating universes

If you are following this chapter sequentially, then you still have Ptolemy running from previous sections. To see how Ptolemy will behave when started in your own directory, exit `pigi`. Do this by typing a control-d character in the `vem` console window. A dialog box may appear with a menu of facets that `vem` thinks have been changed. Since all of these belong to the user "ptolemy", you do not want to save them. If it appears, do not select any of them. Just click "OK". A a warning window may then appear telling you that closing the console window will terminate the program. Just click "Yes".

In this section, we will show how to create your own universes with a simple example that is very similar to the `sinMod` demo explored above. First, be sure you are in a directory where you have write permission, like your home directory.

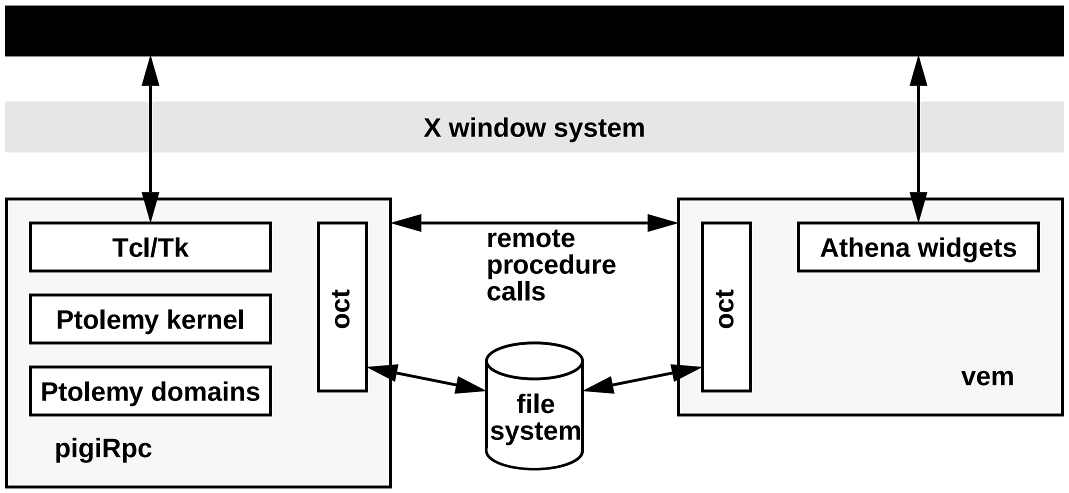


FIGURE 2-6: The software architecture of the Ptolemy design environment running under pigi, the graphical interface. The user interacts with two Unix processes, pigiRpc and vem.

- Create a new work area:

```
mkdir example
cd example
```

- Start `pigi`:

```
pigi
```

You will see the message:

```
creating initial facet "init.pal"
```

Wait until the welcome window with the picture of Ptolemy appears. We are now ready to learn about the basics of using `vem`.

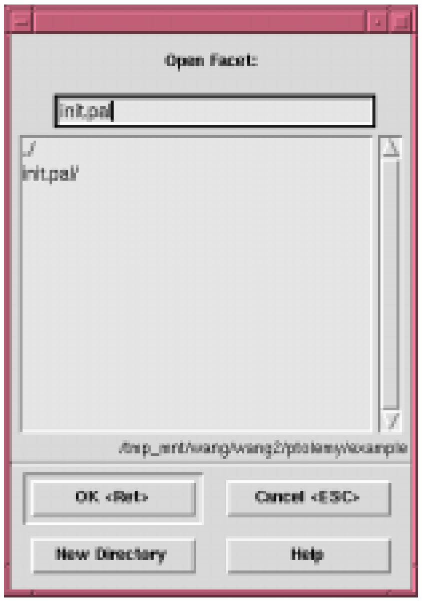
2.7.1 Opening working windows

Now we are ready to create a simple universe. Let’s create a simulation that generates a sine wave and displays it.

- Open a new facet: The facet that is already open, called “`init.pal`”, is special because `pigi` always opens a facet by this name in the directory in which it starts. Convention in Ptolemy dictates that “`init.pal`” should be used to store icons representing complete applications, so instead of using this facet, we will create a new one.

  - Place the cursor in window labeled “init.pal:schematic”.

  - Select the *open-facet* command from the “Window” `pigi` menu (shift-middle-button). Alternatively, type an “F”. You will get a directory browser that looks like this:



  - Replace the name “init.pal” in the text widget with “wave” and click the “OK” button (or hit the return key). A quick way to delete the “init.pal” is using control-u. A new blank window will appear.

- Open a palette:
    - Place the cursor in either blank window.
    - Select the *open-palette* command from the "Window" `pigi` menu. Alternatively, type an "O".
    - `Pigi` will present a palette menu. Select the "sdf" palette by clicking the left button in the box next to "$PTOLEMY/src/domains/sdf/icons/main.pal" (the first entry) and then click on "OK".
    - The palette that opens is shown on the left of figure 2-7. This palette shows the basic categories of synchronous dataflow stars that are available. There are too many stars to put in just one palette. You can use the Window:look-inside ("i") command to open any of the palettes. At this point you should look inside the "Signal Sources", "Nonlinear Functions", and "Signal Sinks". Arrange these palettes on the screen so that you can see the blank window labeled "wave". The stars and palettes are summarized in "An overview of SDF stars" on page 5-4.

2.7.2 Some basic vem commands

At this time, it is worth exploring some basic `vem` commands for manipulating window displays. `Vem` uses post-fix commands. This means that the user enters the arguments to a command before the command name itself. Arguments appear in the `vem` console window as the user enters them. Note that although the text of what the user enters is displayed in the console window, the cursor should be in one of the facet windows.

There are several types of arguments. Each argument type is entered in a different way. All graphics arguments are created with the left mouse button. The five types of arguments are listed below:

Point: Position the cursor, click the left mouse button.

Box: Position the cursor, drag[1] the left mouse button.

Line: Make a point, position the cursor on the point, and drag the left mouse button.

Object: Use *select-objects* and *unselect-objects* commands (explained later).

Text: Enclose text in double quotes.

Arguments can be removed from the command line by typing the delete key, backspace key, or "control-u", which deletes all the current arguments. There are three ways to enter commands:

Menus: Click the middle-button for `vem` commands, shift-middle-but-

1. "Drag" means to press down on a mouse button, move the mouse while holding it down, and then release the button.

ton for `pigi` commands. Menus are of the "walking" variety, as explained before.

Key bindings: Commands can be bound to single keys and activated by just pressing the key. Key bindings are also called "single-key accelerators", and are case sensitive. The key bindings are summarized in table 2-2 on page 2-7 and table 2-3 on page 2-11.

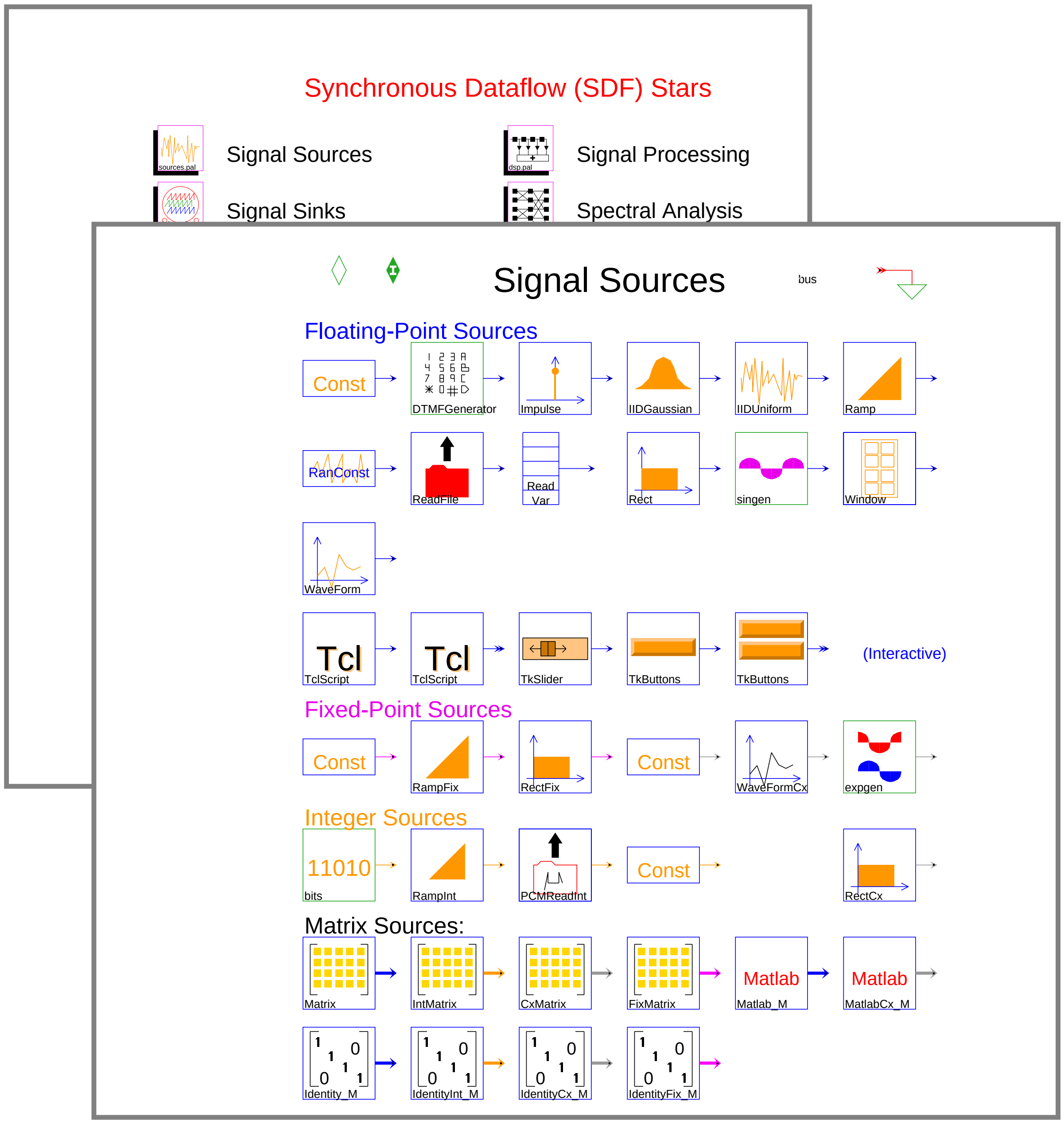


FIGURE 2-7: The master palette for the stars in the SDF domain (left) and one of the sub-palettes (right). The subpalette shows "sources" (signal generators). The palettes are explained in more detail in "An overview of SDF stars" on page 5-4.

Type-in: Type a colon followed by the command name. This is rarely used by Ptolemy users, but `vem` experts use it occasionally.

Let's try a few examples, some of which should be familiar by now. Place the cursor in one of the palette windows (containing library stars) and:

- Type "shift-Z" (capital Z) for *zoom-out*. This makes everything smaller.
- Type "z" (lower-case z) for *zoom-in*. This makes everything bigger. If you zoom in sufficiently, labels will appear below each icon giving the name of the star. `Vem` does not display these labels if they would be too small.
- Try "p" for *pan*. Pan moves the spot under the cursor to the center of the window.
- The `vem` *pan* command can also take as an argument a point which will indicate the new center of the window. Recall that the argument must be entered first. Place a point somewhere in the palette window by clicking the left button, and type "p". The location of your point became the center of the window.
- The `vem` *open-window* command can take a box as an argument. Draw a box in the palette window by dragging the left mouse button and then type "o", or find the *open-window* command in the `vem` menu.
- Try placing points in the new window. Notice that they also appear in the original palette window. Also notice that you are only permitted to place points at certain locations. `Vem` has an implicit *grid* to which points *snap*. The default snap resolution is suitable for making Ptolemy universes.
- You can get rid of your point (or any argument list) by typing "control-u". You can delete arguments one-at-a-time by typing "control-h". Try placing several points and then deleting them one by one.
- You can close the new window (or any `vem` window) with "control-d".
- A particularly useful command at this time is *show-all*, or "f". This rescales and recenters the display so that everything in the facet is visible. Try this command in the palette window that you have been working with.
- You can also resize a window, using whatever X Window bindings you have installed, and then type "f" to rescale the display to fill the window.

2.7.3 Building an example

- Create an instance of the star called "`Ramp`". This star is at the upper right of the sources palette. Its icon has an orange triangle. To do this:
  - Put the cursor in the window "wave:schematic".
  - Create a point anywhere in the window by clicking the left button.
  - Move the cursor over the "`Ramp`" icon in the palette and press the "c" key. This is a key binding that executes the `vem` "create" command.

  - You have just created an **instance** of the “`Ramp`” icon. The actual data that describes how the “`Ramp`” icon should be drawn is stored in another facet (an “interface facet”). An **instance** of the “`Ramp`” icon points to this facet.

- Delete and select instances: Sometimes in the process of editing your schematic, you may need to delete objects. As an example, let’s create another `Ramp` instance and then delete it.

  - Create another `Ramp` instance next to the first one: place a point near the original `Ramp`, place the cursor over the `Ramp` icon in the palette and press “c”. Actually, you don’t have to use the icon in the palette — you could also put the cursor over the already existing `Ramp` icon to achieve the same effect.

  - Place the cursor over the new `Ramp` icon and execute *select-objects* by typing “s”. This creates an object argument on the `vem` command line. This is necessary because the `vem` *delete-objects* command takes arguments of type “object”. The *select-objects* command takes point, box, and/or line arguments and turns the items underneath them into object arguments. The *unselect-objects* command (“u”) does the reverse of *select-objects*.

  - Execute *delete-objects* by typing “D” (upper-case!). This deletes the objects we selected previously.

  - You could also have deleted the newly created Ramp with the *undo* command (“U”). This is an infinite undo, so you can backtrack through all changes you have made since starting the `vem` session by repeatedly executing the undo command.

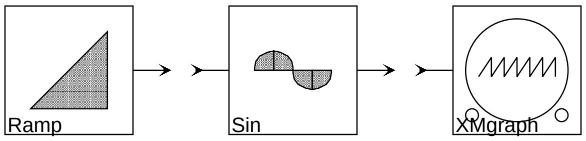
  - Occasionally when you use the select and unselect commands, the objects are not redrawn correctly. When this happens, use the `vem` *redraw-window* command, “control-l” (lower case L), to redraw.

- Create the remaining instances in our example:

  - Create an instance of the “`Sin`” icon to the right of the Ramp. “`Sin`” is in the “nonlinear” palette, where icons are arranged alphabetically by name. Make sure it does not overlap with the Ramp icon. If it overlaps, you can delete it and create a new one.

  - Create an “`XMgraph`” instance to the right of the `Sin` icon. “`XMgraph`” is the first icon in the first row of the “sinks” palette.

We now have three icons: a `Ramp`, a `Sin`, and an `XMgraph`. Your facet should look something like this:



Next, we will connect them together.

- Connect the `Ramp` output to the `Sin` input using the following steps:

  - With the mouse cursor in the “wave” window, type “f” to show all. This will rescale your system, and make it easier to make connections.

  - Draw a line between the output of the `Ramp` and the input of the `Sin`: put the cursor over the `Ramp` output, double-click on the left mouse button, drag the cursor to the `Sin` input, and then let up on the mouse button. If the two terminals are not on a horizontal line, you can bend the line by momentarily releasing the mouse button while dragging it.

  - Type “c” (for *create*) to create a wire. Notice that the *create* command creates wires or instances depending on the type of arguments it is called with.

  - If you need to delete a wire, you can draw a box around it (click and drag with the mouse), select it (press “s”), and then delete it (“D”).

- Connect the `Sin` output to the `XMgraph` input in a similar way.
- Run the universe: We now have a complete universe that we can simulate.

  - Execute the *run* command from the `pigi` “Exec” menu, or type an “R”.

  - Enter “100” for “When to stop”. Do this by typing “control-u” to remove the default entry in the text widget and typing 100. This specifies that the system should be run through 100 “iterations”. What constitutes an iteration is explained in chapter 5, “SDF Domain” on page 5-1. For this simple system, it is just the number of samples processed.

  - Clicking on the GO button or typing a return character will run the system.

A new window with a graph of a rough sine wave should appear. The system generates the sine wave by taking the sine of a sequence of increasing numbers generated by the `Ramp` star. The execution of the `XMgraph` star created this new window to show the output of our simulation. To remove this window, click on the “Close” button or press “control-d” in it.

- Save the facet by typing “S” (upper-case) with the cursor in the “wave” window. This executes the `vem` *save-window* command. It is wise to periodically save your work in case the editor or computer system fails unexpectedly.

- Change parameters: If we look at the output, the sine wave appears jagged. This is because the `Ramp` star has a set of default parameters which cause it to generate output values with an increment that is too large. We can change the parameters of as follows:

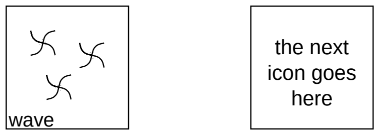
  - Place the cursor over the `Ramp` icon and execute *edit-params* in the `pigi` menu (or type “e”). A dialog box will appear that shows the current parameters.

  - Replace the value of *step* with “PI/50”. (You can use “control-u” to erase the old value.) Finally, click the “OK” button to store the new parameters. This is an example of Ptolemy’s parameter expression syntax, explained above.

- Run the simulation again using 100 iterations. This time the output should look like one cycle of a reasonably smooth sine wave.
- Use *save-window* again to save the new parameters.

To be able to conveniently access this example again, you should create an icon for it. We will do this with the `pigi` command “Extend:make-schem-icon”, or “@”.

- Place the mouse cursor in the “wave” facet window, and hit the “@” key. A dialog box appears asking for the name of the palette in which you would like to put the icon. By convention, we put universe icons in palettes called “init.pal”. So replace the default entry (which should be “./user.pal”) with “init.pal”. When the icon is made, find the “init.pal” window that first opened when you started the system, and type “f” to show all. It should look like this:



  Looking inside this icon (“i”) will get you your “wave” facet. The second item in the palette is a marker indicating where the next icon that you create will go. Henceforth, anytime you start `pigi` in this same directory, the first window you will see will be this “init.pal” window.

- Our example is now complete. To exit:

  - Close all pxgraph windows with “control-d”.

  - Type “control-d” in the `vem` console window. If nothing needed to be saved, the program exits immediately. Otherwise, a dialog box appears asking you to choose buffers to be saved. Unfortunately, as of this writing, some of the buffers listed may have already been saved and do not need to be saved again. The program is overly cautious. To indicate which of the listed buffers you wish to save, click on the box to the left of each name. Then click on the “OK” button.

- A final warning appears telling you that closing the console window will terminate the program. Click on “Yes”.

2.8 Using galaxies

In this section we will explain how to create galaxies. Galaxies allow you to use hierarchy to partition your design into more manageable pieces and to re-use designs as components in other designs.

2.8.1 Creating a galaxy

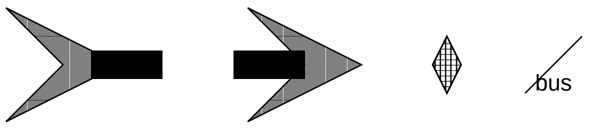
Use the schematic we created in the last example to make a sine wave generator galaxy.

- Instead of modifying our previous example, we will make a copy of it. In your “example” directory, type:

```
cp -r wave singen
```

The recursive copy, `cp -r`, is necessary because `oct` stores data using a hierarchical directory structure. Of course, if the facet `singen` exists already, you must remove it with `rm -r` first before copying.

- Start `pigi`.
- Use *open-palette* (or “O”) to open the “$PTOLEMY/lib/colors/ptolemy/system” palette (the last one in the list of palettes). The system palette contains input and output ports which can be instantiated into schematics just like stars. The contents of the palette are shown below:



- Use *open-facet* (or “F”) to open the “singen” you created using “`cp -r`”. You can use the file browser shown on page 2-23; just double click on the name “singen” in the lower window of the browser.
- In the “singen” window, delete the `XMgraph` star and the wire attached to it. The easiest way to do this is draw a box (click-drag) around the star and its input wire, press “s” to select these objects, and then press “D” to delete them. (You may want to enlarge your window to make it easier to work.)
- Place an output port where the `XMgraph` star used to be and connect it to the output of the `Sin` star. The output port is the icon in the system palette with an arrowhead (an input port, by contrast, has a fish tail), as shown above.
- Name the output port “out”:
    - Position the cursor over the black box on the new output port.

- Type “out” as a text argument, including quotation marks.

- Type “c” for *create*. Again, note that the *create* command has a different action than before. It names input or output terminals when given a text argument.

We now have a galaxy. The fact that a schematic has input or output ports distinguishes it as a galaxy. This galaxy that you just created is similar to the “singen” galaxy in the “Signal Sources” palette. Find it, and look inside, to make the comparison.

2.8.2 Using a galaxy

We have just created a galaxy that we would like to use in another design. In order to do this, we need to create an icon for this galaxy that we will then instantiate in our other design.

- Create an icon:

  - Place cursor in “singen:schematic” window.

  - Execute *make-schem-icon* in the `pigi` “Extend” menu (“@”).

  - The dialog box should contain:
    ```
    Palette: ./user.pal
    ```
    This specifies the name of the palette that will contain your icon. By convention, we usually put galaxy icons in the palette called “user.pal” in the current directory. Hence, this is the default name.

  - Since you had already created an icon for the “wave” universe, and that icon was copied by your `cp -r`, `vem` asks whether it is OK to overwrite the icon. Click “OK”. Wait until *make-schem-icon* is done. `Vem` informs you that it is done with a message in the `vem` console window, which may be buried by now.

- Open the palette called “`user.pal`” using *open-palette* (“O”). The newly created galaxy icon should appear in this palette along with the same special icon we saw before, called a **cursor**. A **cursor** distinguishes a palette from other types of facets and it determines where the next icon will be placed.
- At this point, we have an icon for our sine wave generator galaxy. It is this icon facet that is instantiated in the “`user.pal`” palette. We can now use our sine generator galaxy simply by instantiating our icon into another schematic.
- Use *open-facet* to create a new facet with the name “`modulation`”.
- In the “modulation” window, create a universe that takes two “singen” galaxies, multiplies their outputs together, and then plots the result using “`XMgraph`”. The “`XMgraph`” star can be found again in the “Signal Sinks” subpalette of the SDF palette. The multiplier star, called `Mpy`, is in the “Arithmetic” subpalette.

Hint: If you place the icons so that their terminals fall on top of one another, then a connection gets made without having to draw a wire.

- Now run the universe with “when to stop” set to 100. The output should appear as a squared sine wave, which is just a sine wave of twice the frequency shifted up by 1/2.

So far, we have created a galaxy and used it in another universe. But we could also have used our galaxy within another galaxy. In this way, large systems can be broken up into smaller more manageable pieces.

2.8.3 Galaxy and universe parameters

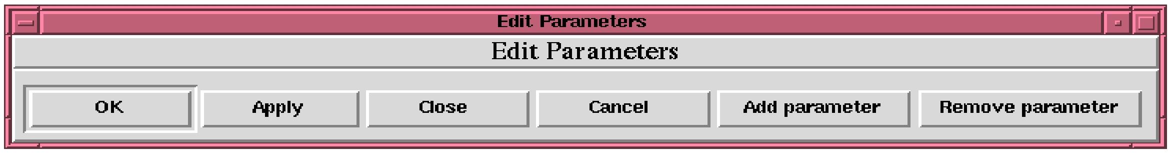
One of the problems with the “singen” galaxy that we just created is that it generates sine waves with a fixed frequency. We would like to make the frequency of the generator parameterizable. That way, we could set the two “singen” galaxies in our “modulation” universe to two different frequencies.

To make a galaxy parameterizable, we create **formal parameters** in the galaxy and then link the formal parameters to the **actual parameters** of the instances contained in the galaxy. The terms, “formal” and “actual” parameters, are analogous to formal and actual parameters in any procedural programming language. An example will make this clear.

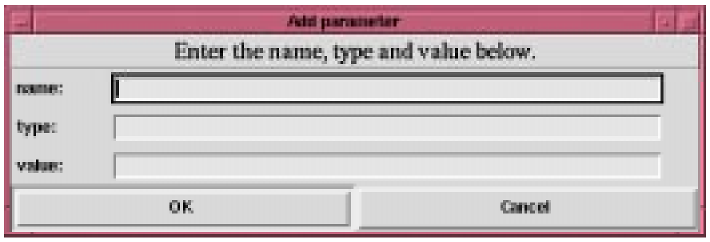
- Create formal parameters:

  - Place the cursor in the “singen” window but away from any instance, i.e., in the grey background of the facet.

  - Execute *edit-params* (“e”). An empty parameter window will appear, looking like this:



To add parameters to the galaxy, click on the “Add parameter” button. A window appears looking like this:



Fill in the dialog as follows:

```
name: freq
type: float
value: PI/50
```

The value will be the *default* value. Then click on “OK”. Recall that you can use “tab” to move from one field to the next of the dialog box and “Return” instead of “OK”. Hence, the dialog can be managed from the keyboard without requiring the mouse.

We just created a new formal parameter called “freq” with a default value of “PI/50”. Additional parameters may be added or old ones changed. The default value of a formal parameter can always be changed by executing *edit-params* in the background of the galaxy. Executing edit-params on the icon representing the galaxy changes the parameter values only for the instance represented by the icon. It overrides the default value specified in the background of the galaxy definition. The possible types for parameters are listed in table 2-6. The syntax for specifying values for parameters is described above in “Changing or setting parameters” on page 2-15. Exactly the same procedure can be used to attach formal parameters to a universe. This allows you to parameterize a complete Ptolemy application.

- Link formal parameters to actual parameters:
  - Place the cursor over the `Ramp` icon in the `singen` window.
  - Execute *edit-params* and fill in the dialog as follows:
    ```
    step: freq
    value: 0.0
    ```
    This allows the *freq* parameter of a singen instance to control the increment of the internal ramp star, thus controlling the output frequency.
- Change the frequency of one of the `singen` instances to “PI/5” by using *edit-params*. This `singen` will be ten times the frequency of the other.
- Run the universe with an iteration of 100. The output should show the product of two sine waves with different frequencies. Don’t forget to save the facets we just created. It

Type name	Description	Example
float	floating-point number	0.2/PI
int	integer	10
complex	pair specified as (real-part, imag-part)	(1.0, 2.0)
string	string	this is a string
floatarray	array of floating-point numbers	0.0 [10] 1.0 0.0 [10]
intarray	array of integers	1 2 3 4 5 6 7
complexarray	array of complex numbers	(0.1, 0.2) (0.3, 0.4) (0.5, 0.6)
stringarray	array of strings	this string array “has five” elements
file	filename	/tmp/input.test

TABLE 2-6: Parameter types supported in Ptolemy

would also be a good idea to create an icon for `modulation` and put it in `init.pal`.

2.9 Editing Icons

`Pigi` automatically generates icons for stars and galaxies, respectively, when you invoke the *make-star* or *make-schem-icon* command from the Extend menu. `Pigi` puts the new icon in a user-specified palette, which by default is user.pal in the directory in which you started `pigi`. More or less any `vem` manipulations can be performed on this icon, but some guidelines should be followed. These icons have a generic symbol, shown in figure 2-2 on page 2-5. To change it, place the cursor over the icon and execute the *edit-icon* ("I") command in the `pigi` menu.[1] A new window containing the icon facet will appear.

Recall from section 2.6 on page 2-21 that icons are stored in *interface facets* and that the icons that appear in *contents facets* are really instances of icons. These instances merely refer to the actual icon facet. The *edit-icon* command opens a window into the actual icon. Any changes made in this window will affect the appearance of all instances referring to the icon.

Recall also that icon facets store a different kind of data from other facets. Icon facets contain information that tells `vem` how to draw objects. Hence, a different set of commands must be used to edit icons. Whenever you edit an icon, `vem` switches to a different mode called "physical editing style." In this mode, we create objects such as lines, boxes, and polygons. This is in contrast to "schematic editing style" which we used before to create instances and connect them together with wires. Physical editing style shares many commands with schematic editing style. For example, *select-objects* is active in both modes. A list of useful physical editing style commands and their key bindings is given in table 2-7.

The commands that create geometry expect a layer argument. The layer of an object determines its color and its fill pattern. To specify a layer, place the cursor over an object attached to the desired layer before executing a command. You can open a palette of layers with the *palette* ("P") command. The palette is shown in figure 2-8.

The layer palette contains several columns of solid and outline colors, with the name of the color at the top of the column. Colors at the top of each icon will be layered on top of colors below them in the columns. A set of special layers are arranged at the bottom of the palette. The layers for icon stems are explained below. The layers for icon bodies define the icon background and optional icon shadow.

A few simple notes will help greatly. First note that when the icon window is opened, the snap is automatically set to 5 "`oct` units". This is because the default snap for schematic windows, normally 25 units, is far too coarse for most icon editing functions. A reasonable compromise is 5 units, unless you are going to try to create a very elaborate icon, in which case 1 unit is probably what you want. The `vem` Options:window-options command allows you to change the snap.

When editing an icon, the `vem` menu is slightly different than when you are editing a schematic. In `vem` terminology, this is because you are working with the *physical view* of a facet. The commands are shown in table 2-7. Most icons can be created by experimenting with the following operations:

1. You must have write permission on the facet to change the icon.

• Select the default symbol within the icon that Ptolemy created when it created the icon

Menu	Heading	Command	Key	Description
vem	none	no command name	cntr-h	remove the last argument (point, box, etc.)
			del	remove the last argument (point, box, etc.)
			cntr-u	remove all arguments from the argument list
			cntr-l	(control lower case L) redraw the window
	System	open-window	o	open a new view into a facet
		close-window	cntr-d	close a window
		where	?	find the position of the cursor in oct units
		palette	P	open the color palette for editing icons
		save-window	S	save a facet
		bindings	b	display key bindings (single key accelerators)
		re-read		restore a facet to the last saved version
	Display	pan	p	move the view to be centered at a given spot
		zoom-in	z	zoom in for a closer view of a facet
		zoom-out	Z	zoom out
		show-all	f	rescale the schematic to fit the window
		same-scale	=	used to get two windows to use the same scale
	Options	window-options		adjust snap, grid spacing, etc.
		layer-display		selectively display colors
		toggle-grid	g	turn on or off the grid display
	Undo	undo	U	undo any number of previous changes
	Edit	create-geometry	c	create a line, box, circle, etc.
		alter-geometry	a	replace an object with one on the argument list
		change-layer	l	change the color of an object
		set-path-width	w	change the width of lines
		create-circle	C	draw a filled or empty circle
		edit-label	E	specify or modify a label
		delete-physical	D	remove the specified object
	Selection	select-objects	s	add an object to the argument list for a command
		unselect-objects	u	remove an object from the argument list
		move-physical	m	move an object
		copy-physical	x	copy one or more objects in a schematic
		transform	t	rotate or reflect an object
		select-terms	cntr-t	select terminals
		delete-physical	D	delete objects
	Application	rpc-any	r	start a vem application (pigiRpc is one)

TABLE 2-7: A summary of the Ptolemy commands in the vem menu in icon editing mode. These commands are obtained by clicking the middle mouse button without holding the shift button when your mouse cursor is in an icon window. The single-key accelerators for commands that have them are shown. More complete documentation can be found in chapter 19, “Vem — The Graphical Editor for Oct” on page 19-1. The command that differ significantly from those in table 2-3 are shaded.

(a star, galaxy, cluster of galaxies, or palette symbol, as shown in figure 2-2 on page 2-5). You can do this by drawing a box (drag the left mouse button) and typing “s” (or using Selection:select-objects in the menu). You can unselect with “u” or control-u. An alternative selection method is to place a point and type “s”. This usually provokes a dialog box to resolve ambiguities. Delete whatever parts of the icon you don’t want using “D” or Selection:delete-objects. WARNING: Do not delete terminals! If you accidentally delete a terminal, the easiest action is begin again from scratch, asking `pigi` to create a new icon.

- Bring up the `vem` color palette by typing “P” (or System:palette in the menu). You will get the window shown in figure 2-8.
- Draw a line by clicking the left button to place a point, and pushing and dragging the mouse button from the same point. Then move the mouse to desired color in the color palette and type “c” (or Edit:create-geometry from the menu). A line may consist of multiple line segments by just repeatedly pushing and dragging the mouse button.
- To create filled polygons, place points at the vertices, then type “c” on the appropriate solid color in the palette.
- To create a circle, place a point at the center, a point on periphery, and type “C” on the appropriate color. To create a filled circle, use a line segment instead of a pair of points.
- Objects can be moved by selecting them, dragging the mouse (using the right button) to produce an image of the object in the desired place, and typing “m”.
- You may change (or delete) the label that `pigi` automatically puts at the bottom of the icon. To change it, select it and type “E” (Edit:edit-label in the menu). The resulting

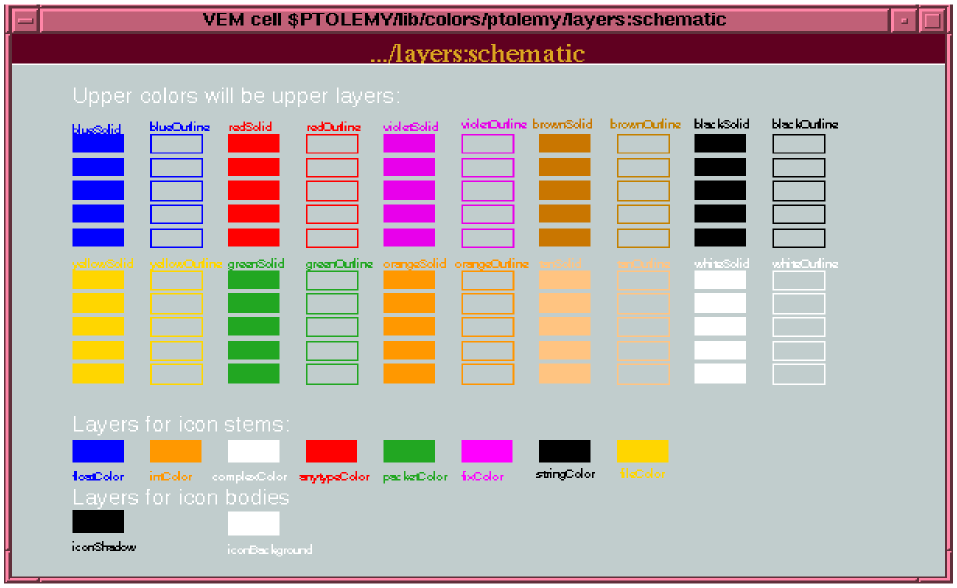


FIGURE 2-8: The palette of colors and layers that can be used to create icons. This palette is invoked by the “palette” vem command (“P”). Each color has a column of boxes. The higher the box you use, the closer to the front an object will be. The colors at the bottom are special, in that they are associated with particular data types.

dialog box is self explanatory. The standard Emacs-like editing commands apply.

- BE SURE TO SAVE YOUR ICON. This can be done by typing “S” (System:save-window in the `vem` menu). You can close your window with control-d. Note that `vem` buffers the data in the window. You can close it and reopen it without saving it, as long as the session has not been interrupted.

By convention, the data types supported by a terminal are indicated by the color of the stem that connects the terminal to the body of the icon. The following colors are currently in use:

`ANYTYPE:`	red
`FLOAT:`	blue
`INT:`	brown
`FIX:`	purple
`COMPLEX:`	white
`PACKET:`	green
`FILE:`	yellow
`STRING:`	black

The color is currently set automatically by the icon generator by using layers defined specifically for this purpose, called `anytypeColor`, `floatColor`, `intColor`, `fixColor`, `complexColor`, `packetColor`, `stringColor` and `fileColor`. These colors are shown at the bottom of the color palette in figure 2-8.

You can change the color of an object manually, if you wish. To do this, select the object, type “"xxxx"”, where xxxx is replaced by the color name (the quotation marks are necessary), and then type the single character “l” (an el — or Edit:change-layer in the menu). Be sure not to change the color of a terminal! Again, be sure to save the window.

One final editing operation is a little trickier: moving terminals. `Pigi` places terminals rather arbitrarily, since it knows nothing of their function. You may wish to have a smaller icon than the default, in which case you have to move the input terminals closer to the output terminals. Or may wish to change the order of the terminals, or you may want to have terminals on the top or bottom of the icon rather than right or left. All of these can be done, but the following cautions must be observed:

- Do not move terminals of icons that have already been used in applications. Unfortunately, if you do this, the block will become disconnected in all applications where it is used. If you are tempted to move terminals in a commonly used block, consider the tedium of finding all applications (belonging to all users) and reconnecting the block. ONLY MOVE TERMINALS ON BRAND NEW ICONS.
- You must respect the default snap of 25 for schematic windows, and move terminals to a point that falls on a multiple of 25 units. Otherwise, connecting to the terminal will be very difficult. Normally, when the icon window opens, the grid lines are 10 units apart. So you can place terminals any multiple of 2.5 grid lines away from the center.
- To orient terminals so they are aiming up or down, select the terminal, type “t” (or Selection:transform in the menu), then type “m” to move it. Repeatedly typing “t” will continue to transform (rotate) the terminal.

A little more detail on the `oct` internals may be useful if you explore the files that are created by these operations. For make-schem-icon, if the schematic is called *xxx*, then the icon itself is stored in "*xxx*/schematic/interface;". The semicolon is part of the filename (this creates some interesting challenges when manipulating this file in Unix, since the Unix shell interprets the semicolon as a command delimiter). The standard stars that are normally part of the Ptolemy distribution are stored in "`$PTOLEMY/src/domains/`*`dom`*", where *`dom`* is the domain name such as `sdf` or `de`. The icons for the stars are stored in a subdirectory called `icons`, the icons for demo systems in a subdirectory called `demo`, and the source code for the stars are stored in a subdirectory called `stars`. Feel free to explore these directories.

Changing the number of terminals in galaxy icons

Whenever the contents of a galaxy are changed so that the new definition has different I/O ports, the icon must be updated as well. You can do this by calling *make-schem-icon* again to replace the old icon with a new one. `Vem` will not allow you to overwrite the old icon if you have instances of the old icon in any open window (regardless of whether the window is iconified). Hence, you must either close those windows with "control-d" or delete the offending icon before replacing it with a new icon. Note that changing number of terminals will also change their layout, so that connections in existing schematics may no longer be valid.

2.10 Sounds

On some workstations (currently only SGI Indigos, HP 700s and HP 800s and Sun SparcStations,), Ptolemy can play sounds over the workstation speaker. Below we discuss various details about playing sounds on various workstations.

2.10.1 Workstation Audio Internet Resources

Below we list several workstation audio resources on the Internet.

`ftp://ftp.cwi.nl/pub/audio`
: Home of the audio file format FAQ.

`http://orbit.cs.engr.latech.edu/AF`
: The AF program is an audio server similar to the X server which allows remote machine to play audio on the local machine. The user starts the AF program in the background and then uses the `aplay` program to play sounds. AF is not directly supported by Ptolemy, but is nonetheless useful.

`http://www.spies.com/Sox/`
: The `sox` program converts files between various formats.

`ftp://ftp.hyperion.com/WorkMan`
: The `workman` program can play audio CDs on Sun SparcStations.

2.10.2 Solaris

Sun workstations running Solaris2.x can play 8kHz mu law sounds directly through `/dev/audio`. The Solaris2.x `/usr/openwin/bin/audiotool` program can be used to control the record and play volume and the input and output sources. In Ptolemy 0.7 and later, the SDF Play star writes the appropriate `.au` file header.

Most Sun workstations can only play 8 bit u-law audio at 8khz. Sun UltraSparcs can play a range of audio formats: 8 bit u-law, 8 bit A-law and 16 bit linear. UltraSparcs can also play a range of sample rates, including CD (44.1khz) and DAT (48khz).

The Solaris `/usr/demo/SOUND` contains sample sounds and programs. See `/usr/demo/SOUND/bin/soundtool` for a graphical sound program with a slightly different interface. For further information about audio on Sun workstations, see the man pages in `/usr/demo/SOUND/man`, and the man pages for `audioamd`, `audiocs`, `dbri`, `sbpro`, `audio`, and `cdio`.

SparcStation CD-ROM

The `workman` program can play audio CD’s via the Sun SparcStation CD-ROM drive. workman can be configured to use the Solaris `volmgt` program so that when an audio CD is inserted into the drive it is automatically played. Only the Sparc5 and a few obscure Sparc10s can get audio from the CD directly.Most other Sparcs can use a mini jump plug from the headphone jack on the CD-ROM to the line in on the back of the machine. You can then use audiotool to control the inputs and outputs. Look under the `Volume` menu button for the proper controls. It may take a few minutes to adjust the levels appropriately. The `workman` program can be used as an audio source with the CGC Tycho demos (see “Tycho Demos” on page 14-27) to demonstrate the various audio effects.

2.10.3 HPUX

Under HPUX10.x, the `/opt/audio/bin/audio_editor` program can play sounds. Under HPXU9.x, use `/usr/audio/bin/audio_editor`.

2.10.4 Playing Audio over the Network

If you use Ptolemy to create audio files, then you may want to share them with others over the network.There are several ways to play audio over the network, we discuss them below.

Via the Web

Audio files can be placed on HTML pages and played by many HTML browsers over the network. There are many proprietary commercial server packages that allow users to listen to audio via their browser, we do not cover those packages here, instead we discuss two common formats: `.au` and `.wav`. In general, SparcStations can directly play only `.au` files and Windows and Macintosh machines can play both `.au` and `.wav` files. If you use Ptolemy to generate a `.au` file, the file must have a proper header. The SDF `Play` star will generate that header for you.

Under Solaris, you can use the `xplaygizmo` and AudioFile `aplay` programs to play audio files via a browser. To set this up, place the following in the `.mailcap` file in your home directory and restart your browser.

```
audio/*; xplaygizmo -p -q /usr/sww/AF/bin/aplay; stream-buffer-
size=2000
```

On the Macintosh to play the `.au` files under Netscape, you may need to install a sound program. If you are using the “Berkeley Internet Kit”, then you probably already have installed a program `SoundApp` that can play the Sun audio files. However, Netscape may not be configured to use it. You can change this by selecting `General Preferences` from the `Options` menu, and selecting the `Helpers` page within that. Under `ULAW audio`, you should set the file type to `ULAW` and the application program to `SoundApp`.

Java

Java can play `.au` files over the net, but again, these files must have a proper header.

AF

The AudioFile program `AF` is a audio server that allows a user to listen to a sound generated on a remote machine. See the link above for more information.

2.10.5 Ptolemy Sounds

You can try playing sounds with the universe you just created. Replace the `XMgraph` star in `waveform` window with an instance of the `Play` star (second row of the sinks palette, right of center, with a stylized loudspeaker as an icon). Edit the parameters of the `Play` star entering 16000 for the *gain* parameter. (To see details about the `Play` star, execute the “profile” command in the “Other” menu, or type a comma (“,”) with the mouse on the `Play` icon). The SPARCstation’s speaker is driven by a codec that operates at an 8 kHz sample rate. So running this universe for 40000 samples will produce about 5 seconds of sound. The sound produced by the current parameters is not particularly attractive. Experiment with different parameter values. Try PI/1000 in place of the PI/50.

An interesting variant of this system modulates a chirp instead of a pure sinusoid with a low frequency sinusoid. A chirp is a sinusoid that sweeps over a frequency range. You could replace one of your `sinegen` instances with something that generates a chirp, and again experiment with parameters.

A chirp can be created with three stars: a `Ramp`, an `Integrator` and a `Sin`, connected in series. The *step* parameter of the `Ramp` should be very small, such as 0.0001. With this value, you will hear some aliasing if you create five seconds of sound. The `Integrator` is in the “arithmetic” palette, furthest on the right, and its default parameter values are fine for this purpose. Use the “profile” command (“,”) to read about it. Note that a fourth star, a `Const` (second star in “sources” palette) is needed to set the `Integrator` *reset* input to zero.

In the SDF domain, sound output is collected into a file, and then played out in real time. Another alternative, available in the CGC domain, is to generate the output in real-time. Since the CGC stars have not been optimized for real-time performance, only simple signals can be generated at this time.

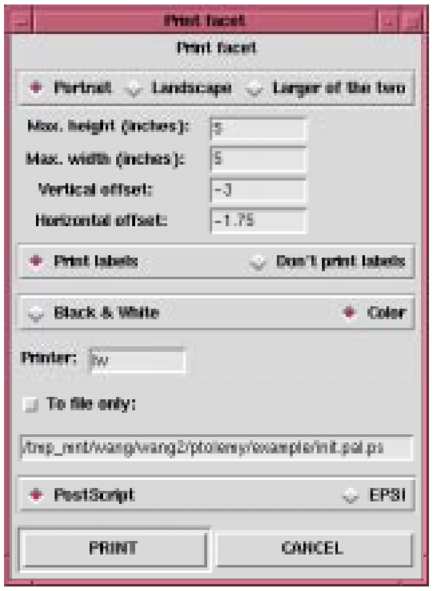
This is a good time to try out your own examples. In general, when you create new galaxies and universes that depend on each other, it is a good idea to keep them together in one directory. For example, all of the facets we have created so far are in the “example” directory. You can use the extensive Ptolemy demos as models.

2.11 Hardcopy

There are several options for printing graphs and schematics developed under Ptolemy. The first option generates a PostScript[1] description and routes it to a printer or file. The second uses the screen capture capability of the X Window system.

2.11.1 Printing oct facets

A block diagram under `pigi` is stored as an `oct` facet. To print it to a PostScript printer, first save the facet. Do this by moving the mouse to the facet and press the "S" key to save. The facet must be saved. Then, keeping the mouse in the facet, invoke the "print facet" command from the "Other" `pigi` menu. You will get a dialog box that looks like this:



Print facet

Print facet

[✓ Portrait] Landscape Larger of the two

Max. height (inches): 5
Max. width (inches): 5
Vertical offset: -3
Horizontal offset: -1.75

[✓ Print labels] Don't print labels

Black & White [✓ Color]

Printer: lw

☐ To file only:

/tmp_mnt/wang/wang2/ptolemy/example/init.pal.ps

[✓ PostScript] EPSI

Most entries are self-explanatory. The default printer is determined by your environment variable PRINTER, which you can set by putting the following line in your .cshrc file:

```
setenv PRINTER printername
```

You will have to restart `pigi` for this change to be registered.

If you select the option "To file only", then PostScript code suitable for importing into other applications will be generated. The image will be positioned at the lower left of the page. The facets displayed in this document were generated this way and imported into FrameMaker. See chapter 18, "Creating Documentation" on page 18-1 for more information.

The "EPSI" option will create Encapsulated Postscript output. Note that you need to have GNU ghostscript installed to generate EPSI. See "Other useful software packages" on page A-14 for further information about GNU ghostscript.

2.11.2 Capturing a screen image

Under the X window system and compatible systems, there are facilities for capturing screen images. These can be used directly with Ptolemy. However, colors that work well on the screen are not always ideal for hardcopy. For this reason, two sets of alternative colors have been devised for use with black and white printers, these color sets are selected at startup with the pigi command line option `-bw` or `-cp`. For black and white printers, use the `-bw` command line option when starting Ptolemy, as in:

1. PostScript is a registered trademark of Adobe Systems Inc.

```
        pigi -bw
```

The screen capture command can be used effectively. For example, under the X Window system, the following command will print a window on a black and white PostScript printer:

```
        xwd | xpr -width 4 -portrait -device ps -gray 4 | lpr
```

If you wish to grab the window manager frame, then you can use:

```
        xwd -frame > myfile.xwd
```

Other alternatives include a program called `xgrabsc` or some equivalent that may be available on your windowing system. A simple use of this is to generate an encapsulated PostScript image using the following command

```
        xgrabsc -eps -page 4x2 -o mySchematic.ps
```

where “mySchematic.ps” is the name of the file into which you would like to store the EPS image. Then with the left mouse button, draw a box around the desired portion of the screen. This command will then save an encapsulated PostScript file four inches by two inches called mySchematic.ps. This file can then be used an a wide variety of document processing systems. To grab an entire window, including whatever borders your window manager provides, use the `xgrabsc -click` option.

Importing an image as PostScript

For example, to include this PostScript in a TeX document, include the command

```
        \include{psfig}
```

in the TeX file and use the commands

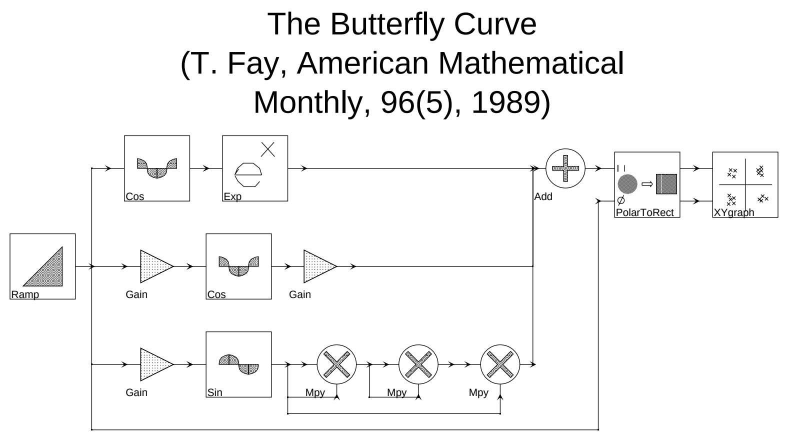
```
        \begin{figure}
             \centerline{
             \psfig{figure=mySchematic.ps,width=4in,height=2in}}
             \caption{Ptolemy Schematic}
        \end{figure}
```

To display the PostScript as a figure within a FrameMaker document.

The “print-facet” command can optionally generate a PostScript file suitable for inclusion in a FrameMaker document. Once you have generated this file, the preferred way to include it is as follows. First, create an anchored frame by using the "Anchored Frame" menu choice under the "Special" menu. The anchored frame will contain two disconnected text columns, one for the figure, the other for the figure paragraph that describes the figure. Create the first disconnected text column using the graphics tools. Then, put in the text box a `#include` line. For example, the text box might contain the following line:

```
        #include /users/ptolemy/doc/users_man/figures/butterfly.ps
```

Unfortunately, the file must be specified using an absolute path, unless you always start FrameMaker from the same directory. With the cursor in the newly created text column, issue the command “Customize Text Frame” from the “Customize Layout” submenu in the “Format” menu, and select “PostScript code.” When you print the document, you will get the following graphic:



The graphic will be anchored at the lower left of the text column you created.

The second disconnected text column is created in a similar fashion with the Graphics tool, but text is entered into the text column rather than the include directive.

It is possible, instead of using the `#include` line as above, to directly import the PostScript file into FrameMaker. However, this makes the text document very large, and the FrameMaker process appears to grow in size uncontrollably. Unfortunately, as of this writing, it does not appear possible to convert these PostScript files to encapsulated PostScript, which would have the advantage of displaying a semblance of the image.

Importing an Image as a X bitmap (XBM)

FrameMaker and some other text processing systems can import and print ordinary color X window dumps. To have these displayed in color on the screen, the following lines may need to appear in your X resources file:

```
        Maker.colorImages: True
        Maker.colorDocs: True
```

One can use various programs, such as the FrameMaker 3.1 utility `fmcolor` and the Poskanzer Bitmap (PBM) tools to reduce a color window dump to a black and white window dump. This will save space and avoid any dithered imitations of color. However, since `fmcolor` applies a threshold based on color intensity to the image, some foreground colors may get mapped to white instead of black. To prevent this, use the `-cp` (`cp` stands for color printer) command line option when starting `pigi`, as in

```
        pigi -cp
```

Then color window dumps can be converted to black and white window dumps using the following FrameMaker 3.1 command:

```
        fmcolor -i 90 filein fileout
```

A useful hint when using such a document editor is to turn off the labels in `pigi` before capturing the image, and then to use the document editor itself to annotate the image. The fonts then will be printer fonts rather than screen fonts. To turn off the labels, execute the `vem` command “layer-display” under the `vem` “Options” menu.

2.12 Other useful information

In this section we cover additional information which may be useful. More advanced topics will be covered in following chapters.

2.12.1 Plotting signals and Fourier transforms

The Ptolemy menu has a submenu called *Utilities* that invokes some useful, frequently used, predefined universes. For example, the "plot-signal" ("~") command will plot a signal. The signal can be read from a file or specified using the syntax for specifying the value of a *floatarray* parameter in `pigi`. For example, if the value of the *signal* parameter is:

```
1 [10] -1 [10]
```

then the "plot-signal" command will plot ten points with value 1.0 and ten points with value -1.0. The *options* parameter can accept any options understood by the pxgraph program (see the pxgraph section of the *Almagest*). To plot a signal stored in a file, simply use the following syntax for the *signal* parameter:

```
< filename
```

You may need to specify the full path name for the file.

Another useful *Utilities* command is *DFT* ("^"), which reads a signal just as above and plots the magnitude and phase of the discrete-time Fourier transform of the signal. These are plotted as a function of frequency normalized to the sampling rate, from $-\pi$ to π. The sampling frequency is assumed to be 2π. A simple phase unwrapping algorithm is used to give more meaningful phase plots. A radix-2 FFT is used, so the order (the number of points) of the fast Fourier transform must be a power of two. That is, the user actually specifies $\log_2(order)$. The order can be longer than the signal, in which case, zero-padding will occur.

2.12.2 Moving objects

Sometimes you may want to move objects around within your schematic. Use the `vem` command *move-objects* ("m") in the *Selection* menu to do this. You can move objects as follows:

- Select the objects that you want to move.
- Using the right mouse button, drag the objects to the desired location.
- Execute *move-objects*, "m".

2.12.3 Copying objects

You can create a new instance of any object in a facet by placing a point where you want the new instance, moving the mouse to the object you wish to copy, and executing "create" ("c"). However, this does not copy the parameter values. If you wish to create a new instance of a star or galaxy that has exactly the same parameter values as an existing instance, you should use the *copy-objects* ("x") command in the `vem` "Selection" menu. To do this, first select the object or objects you wish to copy. Then place a point in the center of the object. Then place a second point in the location where you would like the new object, and type "x". The new object starts life selected, so you can immediately move it, or type "control-u" to unselect it. As of this writing, `vem` unfortunately does not allow you to copy objects from one facet to another.

2.12.4 Labeling a design

It is often useful to annotate a block diagram with titles and comments. The `vem` *edit-label* (“E”) command in the “Edit” menu will do this. It takes two arguments: a point specifying the position of the label, and the name of a *layer*, which determines the color of the label. Place a point where you would like the label, and then type a layer name, such as “blackSolid” (with the quotation marks). Then type “E”. An Athena widget dialog box like that on page 2-14 will appear, offering various options. Type the text for your label in the “Label” box. It can contain carriage returns to get more than one line of text. To select a text height (font size) you can move the slider to the right of the “Text Height” box. The middle button moves the slider by large amounts, and the left and right buttons are used for fine tuning. The initial default is 40, in releases earlier than Ptolemy 0.6, the default was 100, which was too big for all but the loudest titles. Sizes 60 and 40 work well with the overall scaling of Ptolemy facets. You can also change the justification by clicking the left button to the right of each justification box. A pop-up menu lists the options. The colors recommended for labels are:

```
blackSolid
blueSolid
brownSolid
greenSolid
orangeSolid
redSolid
violetSolid
whiteSolid
yellowSolid
```

2.12.5 Icon orientation

Most Ptolemy icons have inputs coming in from the left and outputs going out to the right. To get better looking diagrams, you may sometimes wish to reorient the icons. This can be done with the `vem` command “transform” (“t”). Select the icon you wish to transform and type “t” as many times as necessary to get the desired orientation. Each time, you get a 90 degree rotation. Then execute the move-object “m” to commit the change. Notice that a 180 degree rotation results in an upside down icon. To avoid this, reflect the icon rather than rotating it. To reflect it in the vertical direction (exchanging what’s on top for what’s on the bottom), select the object, type “my” (include the quotation marks), type “t” to transform, and “m” to move. To reflect along the horizontal direction, use “mx” instead of “my”. In summary:

To reflect an object horizontally, select it, and type:

```
"mx" t m
```

To reflect it vertically, type:

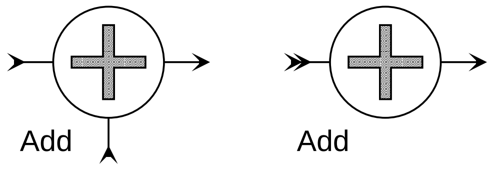
```
"my" t m
```

2.12.6 Finding the names of terminals

Some stars have several terminals, each with a different function. The documentation may refer to these terminals only by name. Unfortunately, the name of a terminal is not normally visible when an icon is viewed with normal scaling. However, zooming in will eventually reveal the name. The easiest way to do this is to draw a box around the terminal and open a new window with the "o" command. Then you can zoom in if necessary. Future versions of Ptolemy will hopefully have a better mechanism.

2.12.7 Multiple inputs and outputs

Ptolemy supports star definitions that do not specify how many inputs or outputs there are. The `Add` and `Fork` stars are defined this way, for instance. Consider the following two icons, found in the "arithmetic" palette of the SDF domain:



They both represent exactly the same star, as you can verify with the "look-inside" command. The icon on the right, however, has a peculiar double arrow at its input. This is a "multiple input" terminal that allows you to connect any number of signals to it. All the signals will be added. The icon on the left has two ordinary input terminals. It can add only two signals. Why have both kinds?

Sometimes, multiple input terminals are not convenient. A rather technical reason is given below, in the section "Auto-forking" on page 2-48. A more mundane reason is simply that schematics often look better with two-input adders.

There are three ways to work with a star that has a multiple-input or multiple-output connection (technically, a "multiporthole").

First, you can just draw multiple connections to or from the double-arrow porthole icon. This is easy, but it has some limitations. You can't control what order the connections will actually be made in. That doesn't matter for an `Add` star, but for some star types it's important to know which connection corresponds to which element of the multiporthole. Also, the connected portholes can't be connected to any other stars, nor can you use delay icons, because `vem` will get confused (see "Auto-forking" on page 2-48).

Second, you can attach a "bus create" or "bus break out" icon to the multiporthole terminal, choosing one that provides the right number of terminals for your schematic. (These icons are available in the "Higher order functions" section of the domain's palette.) This solves both of the problems with multiple connections to a single terminal. It may not make for a very pretty schematic, however.

Third, you can make a custom icon for the star that replaces the double-arrow terminal with the right number of simple terminals. This is what the two-input `Add` icon actually is.This method takes the most work but may be worth it to make the nicest-looking schematic.

Let's go through an example of how to create a star icon that has multiple input terminals based on an existing Ptolemy star that supports multiple inputs. Suppose you need an icon for an adder with eight inputs. As of this writing, unfortunately, you need to have write permission in the directory in which Ptolemy icons are stored to create this new icon. Alternatively, you can create your own version of the star in your own directory (see the programmer's manual). If you have write permission in the directory where the icons are stored, then you can create a new icon with eight inputs as follows.

In any facet, execute the `pigi` command “Extend:make-star” (“*”). A dialog box appears. Enter “Add.input=8” for the star name, “SDF” for the domain, and “$PTOLEMY/src/domains/sdf/stars” for the star src directory (assuming this is where the source code is stored). Note that “input” is the name of the particular multiple input that we want to specify. If you do not know where the source code is stored, then just look-inside (“i”) an existing instance of the star. The `vem` console window and the header of the editing window that open both tell where the star source code is. A second dialog box appears asking you where you would like the icon put. Accept the default, “./user.pal”. Then open user.pal using “O” to see the new eight-input adder icon. You may edit this icon, as explained in “Editing Icons” on page 2-34.

2.12.8 Using delays

In several domains, delays can be placed on arcs. A delay is not a star, but rather is a property of the arc connecting two stars. The interpretation of the delay in the dataflow domains (SDF, DDF, BDF, and most code generation domains) is as an initial particle on the arc. An initial particle for the scalar data types is one whose value is zero. When the arc passes particles containing “message” type data, a delay on the arc will create an “empty” message. Most often, the destination star of the arc must be able to interpret such “empty” messages explicitly in context of the user-defined type because a “zero” might have different meanings depending on the type. Any feedback loop in the SDF domain must have a delay, or the computation in the loop would not be able to begin.

To use these delays in `pigi`, the user places a delay icon on top of the wire connecting two instances. The delay icon is a white diamond with a green border in the SDF and system palettes. You can specify the number of delays by executing *edit-params* with the cursor on top of the delay icon.

Other domains (besides dataflow) also use delays, but the meaning can be quite different. See the appropriate chapter describing the domain.

A new feature added to Ptolemy releases greater than 0.5 is the support of initializable delays for simulation domains. These delays use a different icon from the old white diamond with green borders. The new delays use an icon that is a green diamond with a white border and has an “I” in the middle of the diamond to signify that it is initializable. We have kept around the old delays for backward compatibility, but the syntax for the two is quite different and the user should probably use just one type to prevent confusion.

The syntax for the new delays is that the arguments to the delay are the initial value themselves. There is no value in the argument that signifies the number of delay particles. Instead, a count of the number of values in the delay arguments is the number of delay particles that will be added to the buffer of the arc corresponding to the delay. These arguments are specified as a string and are parsed according to the data type associated to the arc. For example, an initializable delay with parameter “1 0 1” on an arc passing float particles will have a buffer with three initial particles. The three particles will have the values 1.0, 0.0, and 1.0 respectively. If the arc was working on complex particles instead, an error would be given since complex numbers must be specified using a pair of numbers. A proper argument list for the delay in that case would be “(1,0) (0,0) (1,1)”. The shorthand for declaring multiple values in the argument list is valid, just as in the arraystate case. For example, an argument list of “2

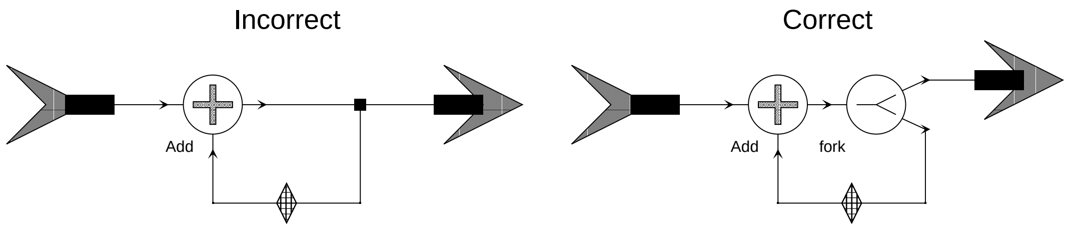
[5]" would specify five initial particles with value 2.

Initializable delays also work on arcs which handle matrix particles. The argument string in this case is parsed differently than above. The first two values in the last specify the number of rows and columns in the initial matrix, respectively. For example, an initializable delay with parameter "1 2 3 [2]" on an arc passing integer matrices would place one matrix with dimension one row by two columns, whose entries all have the value three, in the buffer for that arc. For the case where multiple initial matrices are desired, simply give enough entries in the delay argument string to fill multiple numbers of initial matrices of the given size. For example, an initializable delay with parameter "1 2 3 3 4 4 5 5" on an arc passing integer matrices would create three matrices, all of dimension one row by two columns, such that the first initial matrix on the buffer has all entries equal to three, the second has all entries equal to four, and the last matrix has all entries equal to five.

2.12.9 Auto-forking

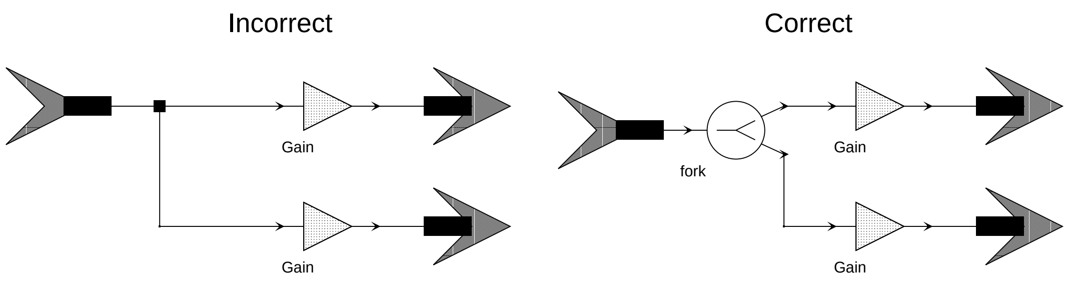
In `pigi`, a single output can be connected to any number of inputs, as one would expect. The interpretation in most domains is that the one output is broadcast to all the inputs. There are several point-to-point connections, therefore, represented by the net.

However, there are restrictions. To understand these restrictions, it is worth explaining that `vem` stores connectivity information in the form of netlists, simply listing all terminals that are connected together. If a delay appears somewhere on a net, and that net has more than one point-to-point connection, then it is not easy to determine for which connection(s) the delay is intended. Consequently, at the time of this writing, delays are disallowed on nets with more than one connection. If you attempt to put a delay on such a net, then when you try to run the system, an error message will be issued, and the offending net will be highlighted. To get rid of the highlighting, execute the `pigi` command "Edit:clear-marks". To fix the problem, delete the offending net, and replace it with one or more `fork` stars and a set of point-to-point connections. An incorrect and correct example are shown below:

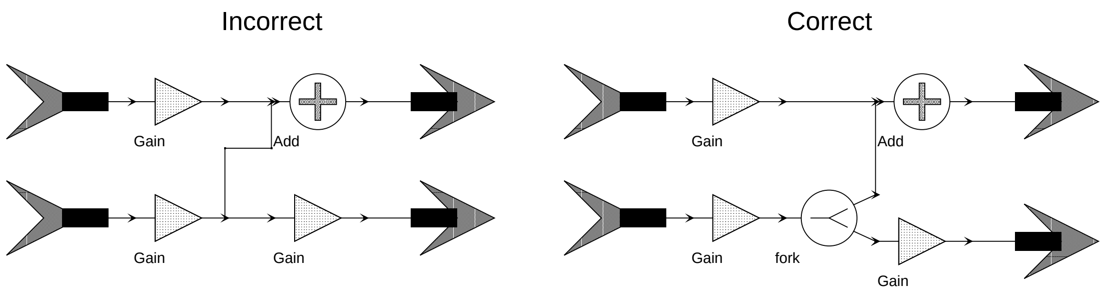


This example also illustrates the use of a delay on a feedback loop. The delay is required here, assuming we are in the SDF domain, because without it, deadlock would ensue. This is due to the fact that the `fork` star cannot fire until the `add` star does, but the `add` star cannot fire until the `fork` star produces its output.

A second restriction is that forks must be explicit when connected directly to input or output terminals of a galaxy. An incorrect and correct example are shown below:



There is also a more subtle restriction. Suppose two outputs are connected to a single multiple-input terminal. Then neither of these outputs can also be connected to some other input terminal. If they are, Ptolemy will issue the error message “multiple output ports found on the same node.” The reason this happens is simple. `Vem` knows nothing about multiple inputs, so it sees a net with more than one output and more than one input. Ptolemy is not given enough information to reconcile this and figure out which outputs should be connected to which inputs. To avoid this problem, it is again necessary to use explicit fork stars, as shown below:



Another solution, which may look nicer than inserting an explicit fork star, is to replace the multiple-input terminal with several simple terminals. You can do that by inserting a “bus create” icon or by using a different icon for the multiple-input star, as was explained in “Multiple inputs and outputs” on page 2-46.

All of the above restrictions may be eliminated in future versions.

2.12.10 Dealing with errors

Ptolemy is composed of several components, as shown in figure 2-6 on page 2-22. When errors occur, it helps to know which component detected the error so that it can be corrected.

When errors occur in `vem`, `vem` prints the error in the console window. For example, if you enter a point argument and execute *create* when the cursor is not over an instance, then `vem` displays the message “Can’t find any instance under spot.” Usually, `vem` errors are easy to fix. In this case, `vem` expects the user to specify the instance to be created.

Errors in the `pigiRpc` process can occur when any of the `pigi` commands are invoked. The error messages, in this case, are displayed in a popup window, which is much more helpful. Error messages may also be displayed in the xterm window in which `pigi` was started. In addition, `pigi` often highlights in red the object in the schematic associated with the error. When this happens, you can execute the *clear-marks* command to clear the highlighting. If such an error occurs and the reason for the error is not obvious, try deleting the indicated objects and redrawing them.

2.12.11 Copying and moving designs

In one of our examples, we used `cp -r` to make a copy of a facet. In general, however, copying entire designs this way does not work. For it to work in the general case, you must also change some data in the facets that you copy. In particular, each facet has pointers to the icons it uses. If you move a galaxy, for example, then any pointer to the icon for that galaxy becomes invalid (or "inconsistent" in `oct` terminology).

A utility program called `masters` has been provided for this purpose. This replaces the program from the `octtools` distribution, called `octmvlib`, that was used with earlier versions of Ptolemy.

Palettes, star icons, galaxies, and universes are stored as `oct` facets. Special care is required when moving or copying `oct` facets. First, as emphasized before, every `oct` facet is stored as a directory tree, so a copy should use `cp -r`. Next, keep in mind that there may be pointers to the moved object in other facets. If you know where all these pointers might be, then moving facets is easy. If you do not know where all the pointers are, then your only practical choice is to leave a symbolic link in place of the old location pointing to the new.

Moving facets

Suppose you have developed a fantastic new galaxy called `alphaCentaur`, and you wish to install it in a directory that is available for general use. Since you have developed the galaxy, you know where it is used. The galaxy icon itself is stored in two facets:

```
alphaCentaur/schematic/contents;
alphaCentaur/schematic/interface;
```

The first of these stores the schematic, the second stores the icon. The peculiar semicolon at the end is actually part of the file name. First move the icon:

```
mv alphaCentaur destinationDirectory
```

This moves the entire directory tree. You must now change all references to the icon so that they reflect the new location. Suppose you have a test universe called `alphaTest`. This should be modified by running the `masters` program as follows:

```
% masters alphaTest
Running masters on wave
Pathname to replace (? for a listing):
```

User input is shown in bold type; program output is shown in regular (not bold) type. Enter a question mark to get a list of all icons referenced in the facet:

```
Pathname to replace (? for a listing): ?
Pathnames currently found in the facet:
     ~yourname/oldDirectory/alphaCentaur
     $PTOLEMY/src/domains/sdf/icons/Ramp
     $PTOLEMY/src/domains/sdf/icons/Sin
     $PTOLEMY/src/domains/sdf/icons/XMgraph
Pathname to replace (? for help):
```

The last three items are pointers to official Ptolemy icons. There is no need to change these. You should now enter the string you need to replace and the replacement value:

```
Pathname to replace (? for help): ~yourname/oldDirectory
New pathname: ~yourname/destinationDirectory
```

Next, use `masters` the same way to modify any palettes that reference the moved icon. For instance, the "user.pal" palette in the directory in which you developed `AlphaCentaur` is a likely candidate. If you miss a reference, `oct` will issue an error message when it tries to open the offending palette, indicating that it is inconsistent.

2.12.12 Environment variables

The following environment variables can be set to customize certain behavior. These should be set (normally) in the user's `.cshrc` file.

`PIGIBW` This variable tells Ptolemy to display all of its windows in black and white.

`PIGIRPC` Specifies an alternative executable file for Ptolemy. Ptolemy is an extensible, modifiable system. Many users will wish to create their own versions to incorporate their own extensions. Details on how to write extensions are given in the programmer's manual, volume 3 of the Almagest. Once you (or someone else) has created a customized version, you can invoke it by specifying the precise name of the executable (complete with its full path, or path relative to an environment variable or user's name). The default executable is `$PTOLEMY/bin.$PTARCH/pigiRpc`. An alternative specification might be:

```
setenv PIGIRPC ~myname/Ptolemy/bin.sol2.5/pigiRpc
```

`PT_DISPLAY` Determines the text editor used to display text files. This determines how text files will be displayed to the user. The value of this variable is a `printf` format string with one `%s` in it. That `%s` is replaced with the name of the file to be viewed. In the default, the `PT_DISPLAY` variable is not set, and the Tycho editor is used.For example, to view files in a new xterm window with the `vi` editor, put the following line in your `.cshrc` file

```
setenv PT_DISPLAY "xterm -e vi %s"
```

and source the file before starting `pigi`.

`PTARCH`	This variable specifies the computer architecture you are using such as `sol2.5`. The architecture setting is returned by the `$PTOLEMY/bin/ptarch` script.
`PT_DEBUG`	If set, this specifies the script to execute when starting pigi in debug mode (using the `-debug` option). An example of a suitable script is `ptgdb`, located in `$PTOLEMY/bin`. This script invokes `gdb`, the Gnu debugger, inside `emacs`.
`PTMATLAB_REMOTE_HOST`	This variable, if set, specifies the name of a remote machine on which to run Matlab if Ptolemy ever invokes Matlab.
`PTOLEMY`	This variable points to the root directory of where Ptolemy is installed.
`PTOLEMY_SYM_TABLE`	This variable is an internal symbol that is used during dynamic linking.
`PTPWD`	This variable gives the command to print the current working directory, which is usually pwd.
`TYCHO`	This variable points to the root directory of where Tycho is installed.

Ptolemy is based on Tcl and [incr Tcl]. These packages set the following environment variables: `TCL_LIBRARY`, `TK_LIBRARY`, `ITCL_LIBRARY`, `ITK_LIBRARY`, and `IWIDGETS_LIBRARY`. See `$PTOLEMY/bin/ptsetup.csh`.

Below we discuss a few Unix system environment variables that affect how Ptolemy functions.

`DISPLAY`	Specifies what X11 Display Ptolemy should start up on. If you are unfamiliar with `$DISPLAY`, then see “Introduction to the X Window System” on page B-1.
`GCC_EXEC_PREFIX` `C_INCLUDE_PATH` `CPLUS_INCLUDE_PATH` `LIBRARY_PATH`	These variables are used by the Gnu compilers to find components of the compilers, see “Gnu Installation” on page A-7.
`HOME`	This variable points to the root directory of the user’s account. This variable must be set for `itclsh` and the software that uses `itclsh` (`ptcl` and `tycho`) to work properly.
`LD_LIBRARY_PATH`	This variable is used by the run time linker to find shared libraries. If you are using prebuilt binaries, and your Ptolemy installation is not at

`/users/ptolemy`, then you may need to set this variable, see "pigi fails to start up, giving shared library messages" on page A-17. See also your Unix `ld` man page, and "Shared Libraries" on page D-1.

`PATH` This variable contains a list of directories of executable programs. The order of the directories listed is very important. See `$PTOLEMY/.cshrc` for guidelines on the proper order.

`PRINTER` Determines the default printer used for hardcopy output. This is used to determine the default printer when printing `vem` facets. If you use the provided makefile to print Ptolemy documentation, then this environment variable will determine the printer used. In `$PTOLEMY/.cshrc` the pertinent line reads:

```
setenv PRINTER lw
```

You should replace `lw` with whatever printer name you are using.

`SHLIB_PATH` Hewlett-Packard systems use `SHLIB_PATH` instead of `LD_LIBRARY_PATH` to find shared libraries. See the `LD_LIBRARY_PATH` description above for details.

`USER` This variable gives the name of the user running Ptolemy. This variable is set by every shell. In your `.cshrc` file, add the following line:

```
if (! $?USER) setenv USER $LOGNAME
```

Many of Ptolemy's domains rely on additional environment variables. The CG56 domain relies on `S56DSP` to indicate the path name where the tools for the S56X Motorola 56000 board are installed and `QCKMON` to indicate the path name where the QCK Monitor tools are installed. The VHDL domain relies on `SYNOPSYS` to indicate the root directory for the installation of Synopsys tools and `SIM_ARCH` to be set to the computer architecture you are using for the Synopsys tools.

2.12.13 Command-line options

The `pigi` program is actually a `csh` script, located in `$PTOLEMY/bin`. That script starts two processes: `vem` and `pigiRpc`. The usage is

`pigi [`*`options`*`] [`*`facet-name`*`]`

The optional facet name specifies a `vem` facet that should be opened upon starting the system. The command-line options are:

`-bw` Use black and white, even on a color monitor. This is useful for generating readable hardcopy from X Window dumps.

`-cp` Fine tune the colors to improve the quality of hardcopy made on a color printer from X Window dumps.

`-console` Open a command console through which the user can issue Tcl commands.

`-debug` Invoke Ptolemy running under `gdb`, a symbolic debugger. If a version of the pigiRpc executable with debug symbols can be found, the script will it. If

`$PIGIRPC` is set then that binary is used. If `$PIGIRPC` is not set, then the program first looks for

```
      $PTOLEMY/bin.$PTARCH/pigiRpc.debug
```

If that is not found, then the program looks for

```
      $PTOLEMY/obj.$PTARCH/pigiRpc/pigiRpc.debug
```

If that is not found, then

```
      $PTOLEMY/bin.$PTARCH/pigiRpc
```

is used. If the `PT_DEBUG` environment variable is set, then its value is the name of a script used to invoke the debugger. For example, the script `ptgdb`, located in `$PTOLEMY/bin`, invokes `gdb` under `emacs`.

`-ptiny` Invoke the smallest version of Ptolemy, if it can be found. The executable that is used is called `pigiRpc.ptiny`. This version contains only the SDF and DE domains, but without the image processing stars and the user-contributed stars.

`-ptrim` Invoke an intermediate-sized version of Ptolemy, if it can be found. The executable used is called `pigiRpc.ptrim`. This version contains only SDF, BDF, DDF, DE, and CGC domains, but without the parallel targets.

`-display` *display-name*
Specify an alternative display to use. If this option is missing, then the `DISPLAY` environment variable is used.

`-help` print out the usage information

`-rpc` *ptolemy-executable*
Specify an alternative Ptolemy executable to use. The default is `$PTOLEMY/bin.$PTARCH/pigiRpc`.

`-xres` *X-resource-filename*
Specify an X resource file to merge before running Ptolemy. The standard X program `xrdb` is used with the `-merge` option.

2.13 X Resources

A large number of X window resources can be set by a user to customize various aspects of the user interface. The best way to explore these is to examine the file `$PTOLEMY/lib/pigiXRes9` for the defaults. These defaults can typically be overridden in the user's.Xdefaults file, and incorporated into the X environment using the program `xrdb`. For example,

```
Vem*font:  *-times-medium-r-normal--*-120-*
```

changes the font in the `vem` console window, menus, dialog boxes, etc., to something smaller than the default. Also,

```
Vem*background: antiqueWhite
```

changes the background in the `vem` console window and dialog boxes to the color "antiqueWhite."

2.14 Tk options

In Tycho, many of the user interface features are controlled through the preferences manager, which is available under the Tycho `Help` menu. In the older non-Tycho Tk windows, a number of user interface options are specified through Tk options rather than directly through X resources. These are defined in the file `$PTOLEMY/lib/tcl/ptkOptions.tcl`. One way to override these is to start pigi with a console window:

```
pigi -console
```

and in the console window, change the options. For example, the command

```
option add Pigi*background gray98
```

changes the dialog box backgrounds to a very light gray. This option was used to create the X window dumps used in this manual.

2.15 Multi-domain universes

The domain of a facet is set using the `pigi` “Edit:edit-domain” or “d” command. This command causes a checklist to appear listing all domains currently linked into the system. All examples in the SDF Demo palette are one-domain applications, using only SDF. Several examples of multi-domain applications can be found in the DDF and DE Demo palettes. It is instructive to explore these applications, using the edit-domain command at all levels of the hierarchy to see what domains are used. In addition, the section “Wormholes” on page 12-4 in the DE chapter contains a useful discussion on mixing the DE domain with other domains in Ptolemy.

Recall that a `Wormhole` in Ptolemy is a block that has a different domain on the outside than on the inside. In `pigi`, wormholes look exactly like galaxies -- in fact, they are both just facets with ports. The only difference is that the domain is different on the inside than on the outside. Thus, whether a particular facet compiles into a plain galaxy or a wormhole depends on whether it is referenced from an outer facet of the same domain or a different domain. You get a wormhole if the domains are different.

To build multi-domain applications, it is necessary to understand the models of computation in each domain, to ensure that application will behave consistently at the domain boundaries. For this, it is necessary to refer to the domain chapters in this user’s manual.

In some domains, it is possible to select one of several targets, which manage the execution of the domain in different ways. The target for a facet is set using the `pigi` “Edit:edit-target” or “T” command. This command causes a checklist to appear listing all targets available for the current domain. If a target is selected (rather than pushing “Cancel”), another dialog box appears containing whatever parameters the selected target may have. Both the current target selection and the parameters for it are recorded with the facet when you execute “save-window”.

If “edit-target” is executed in a galaxy facet (not a universe facet), then it offers a choice labeled `<parent>` in addition to the target(s) for the facet’s domain. This choice simply means “use the outer facet’s target selection and target parameters”. If you select this choice, then no target parameter dialog box appears.

The `<parent>` target choice is extremely important, because . If you choose anything other than `<parent>`, then your galaxy will always be compiled into a wormhole, so that it can have a separate target from the outer galaxy or universe. A wormhole will be created even if you have in fact selected the same domain, same target and same target parameters as in the outer facet --- `pigi` doesn’t check. Thus, if you accidentally set the target choice to something besides `<parent>`, you’ll end up with wormholes rather than plain galaxies. This can cause unexpected behavior, because the semantics of an XXX-in-XXX wormhole aren’t necessarily the same as just embedding a galaxy into another galaxy. (DE domain, in particular, has some oddities with DE-in-DE wormholes as of this writing.) Even if the semantics are unaffected, a wormhole will be slower than a plain galaxy. So be careful to use `<parent>` in galaxies, unless you really intend to create a wormhole having a different target. In most cases, you only want to make specific target selections in universe facets.

Chapter 3. ptcl: The Ptolemy Interpreter

Authors: *Joseph T. Buck*
Wan-Teh Chang
Edward A. Lee

Other Contributors: *Brian L. Evans*
Christopher Hylands

3.1 Introduction

There are two ways to use Ptolemy: as an *interpreter* and a *graphical* user interface. The Ptolemy Tcl interpreter `ptcl` conveniently operates on dumb terminals and other environments where graphical user interfaces may not be available, and is described in this chapter. The Ptolemy graphical user interface `pigi` is described in chapter 2. When `pigi` is run with the `-console` option, a ptcl window will appear. This combination allows the user to interact with Ptolemy using both graphical and textual commands. Invoking `tycho`, the Ptolemy syntax manager also brings up a ptcl interpreter window. To invoke `tycho` from `pigi`, move the mouse over a facet and type a `y`.

In Ptolemy 0.7, the `tysh` binary contains a prototype of a new interface to the kernel called pitcl. If you start `tycho` with the `-pigi`, `-ptrim`, or `-ptiny` options, then you will be running pitcl, not ptcl. Pitcl is not backward compatible with ptcl, and the pitcl interface is bound to change over time. See the Tycho documentation in `$TYCHO/typt/doc/internals/pticl.html` for further information.

The Ptolemy interpreter, `ptcl`, accepts input commands from the keyboard, or from a file, or some combination thereof. It allows the user to set up a new simulation by creating instances of blocks (stars, galaxies, or wormholes), connecting them together, setting the initial values of internal parameters and states, running the simulation, restarting it, etc. It allows simulations to be run in batch mode. We have used batch mode simulation to run regression tests that compare runs from different versions of Ptolemy.

`Ptcl` is based on John Ousterhout's Tcl (tool command language), which is an extensible interpreted language. All the commands of Tcl are available in `ptcl`. This interface is more convenient than the graphical interface when large complex universes are being created automatically by some other program. Some users also find it more convenient when using a symbolic debugger to debug a new piece of code linked to Ptolemy.

`Ptcl` extends the Tcl interpreter language by adding new commands. The underlying grammar and control structure of Tcl are not altered. Commands in Tcl have a simple syntax: a verb followed by arguments. This document will not explain Tcl; please refer to the manual entry at `$PTOLEMY/tcltk/itcl/html/tcl7.6/Tcl.n.html` which is included with the Ptolemy distribution. Two other excellent references on Tcl are books by Ousterhout [Ous94] and Welch [Wel95]. This chapter describes only the extensions to Tcl made by `ptcl`.

3.2 Getting started

Follow the instructions in the section “Setup” on page 2-1. Now type `ptcl` to invoke the Ptolemy interpreter. It is also possible to specify a file of interpreter commands as a command line argument. See “Loading commands from a file” on page 3-13.

3.3 Global information

The interpreter has a *known list* containing all the classes of stars and galaxies it currently knows about. New stars can be added to the known list at run time only by using the incremental linking facility, but this has restrictions (see the `link` command below). You can also make your own copy of the interpreter with your own stars linked in. Galaxies, however, are easy to add to the known list (see the `defgalaxy` command below).

The interpreter also has a *current galaxy*. Normally, this is the most recently defined universe, or the most recent universe specified with the `curuniverse` command. During the execution of a `defgalaxy` command, which defines a galaxy, the current galaxy is set to be the galaxy being defined. After the closing curly brace of the `defgalaxy` command, the current galaxy is reset to the previous current universe.

3.4 Commands for defining the simulation

This section describes commands to build simulations and add stars, galaxies, states, and the connections among them. The commands are summarized in tables 3-1, 3-2 and 3-3.

3.4.1 Creating and deleting universes

The command

```
univlist
```

will return the list of names of universes that currently exist. The command

```
newuniverse ?name? ?dom?
```

creates a new, empty universe named *name* (default “main”) and makes it the current universe with domain *dom* (default current domain). If there was previously a universe with this name, it is deleted. Whatever universe was previously the current universe is not affected, unless it was named *name*. To remove a universe, simply issue the command:

```
deluniverse ?name?
```

If no argument is given, this will delete the current universe. After this, the current universe will be “main.” To find out what the current universe is, issue the command:

```
curuniverse
```

With no arguments, this returns the name of the current universe. With one argument, as in:

```
curuniverse name
```

it will make the current universe name equal to that argument. A universe can be renamed using either syntax below:

```
renameuniv newname
renameuniv oldname newname
```

With one argument, `renameuniv` renames the current universe to `newname`. With two arguments, it renames the universe named `oldname` to `newname`. Note that any existing universe named `newname` is deleted.

3.4.2 Setting the domain

Ptolemy supports multiple simulation domains. Before creating a simulation environment and running it, it is necessary to establish the domain. The interpreter has a *current domain* which is initially the default domain `SDF`. The command

```
domain domain-name
```

changes the current domain; it is only legal when the current galaxy is empty. The argument must be the name of a known domain. The command

command	arguments	description	page
`alias`	*galport b1 p1*	Connect a galaxy port to a block port.	3-6
`animation`	`?on	off?`	Enable or disable printing of star names as they fire.	3-11
`busconnect`	*b1 p1 b2 p2 w ?delay?*	Form a bus connection of width *w* between two multi-portholes.	3-6
`cancelAction`	*action_handle*	Cancel an action previously registered using *registerAction.*	3-15
`cd`	*directory*	Change the current directory to the one given.	3-14
`connect`	*b1 p1 b2 p2 ?delay?*	Form a connection between two portholes.	3-5
`cont`	*?num?*	Continue executing the current universe *num* times (default: 1).	3-10
`curuniverse`	*?name?*	Print or set the name of the current universe.	3-2
`defgalaxy`	*name* { *body* }	Define a new galaxy class.	3-8
`delnode`	*name*	Delete the named node from the current galaxy.	3-12
`delstar`	*name*	Delete the named star from the current galaxy.	3-11
`deluniverse`	*?name?*	Delete the current or named universe.	3-2
`descriptor`	*?block?*	Return the descriptor of *block* (default: current galaxy).	3-9
`disconnect`	*b1 p1*	Remove the connection going to the specified port.	3-11
`domain`	*?name?*	Set the domain, or print the name of the current domain.	3-3
`domains`		List the known domains.	3-3
`exit`		Exit ptcl.	3-15
`halt`		Request that the current simulation stop.	3-10
`help`	*?command?*	Print a short description of *command*, or help on *help* if the argument is omitted.	3-15
`knownlist`	*?domain?*	List the known blocks of *domain* (default: current domain).	3-9

TABLE 3-1: First third of the summary of `ptcl` commands. Arguments are in *italic*; literals are in `Courier`; optional arguments are enclosed in question marks. A block name is indicated by *b1* or *b2* and a port name by *p1* or *p2*.

```
domain
```

returns the current domain. It is possible to create wormholes—interfaces between domains—by including a `domain` command inside a galaxy definition. The command

```
domains
```

lists the domains that are currently linked into the interpreter.

3.4.3 Creating instances of stars and galaxies

The first step in any simulation is to define the blocks (stars and galaxies) to be used in the simulation. The command

```
star name class
```

creates a new instance of a star or galaxy of class *class*, names it *name*, and inserts it into the current galaxy. Any states in the star (or galaxy) are created with their default values. While it is not enforced, the normal naming convention is that *name* begin with a lower case letter and *class* begin with an upper case letter (this makes it easy to distinguish instances of a class

command	arguments	description	page
`link`	*objfile*	Incrementally link *objfile* into ptcl.	3-14
`listobjs`	*class ?name?*	List states, ports, or multiports in the named block (default: current galaxy).	3-9
`matlab`	*command ?arg1? ?arg2?*	Manage a Matlab process and evaluate Matlab commands.	3-16
`mathematica`	*command ?arg1? ?arg2?*	Manage a Mathematica process and evaluate commands.	3-16
`multilink`	*linker_args code.o*	Link arbitrary code into the interpreter.	3-14
`newstate`	*name type value*	Define a state for the current galaxy with a default value.	3-6
`newuniverse`	*?name? ?domain?*	Create a new empty universe (defaults: “main” and the current domain).	3-2
`node`	*name*	Create a node for use by *nodeconnect*.	3-6
`nodeconnect`	*b1 p1 node ?delay?*	Connect a porthole to a specified node.	3-6
`numports`	*b1 p1 number*	Force a multiporthole to have a given number of portholes.	3-7
`permlink`	*linker_args code.o*	Link arbitrary code into the interpreter permanently.	3-14
`pragma`	*b1 b2 name value*	Set pragma *name* to *value* for block *b2* in parent *b1*.	3-12
`pragmaDe-faults`	*target*	print default values of the pragmas for the target	3-12
`print`	*?b1?*	print a description of block *b1* (or the current galaxy)	3-9

TABLE 3-2: Second third of the summary of `ptcl` commands. Arguments are in *italic*; literals are in `Courier`; optional arguments are enclosed in question marks. A block name is indicated by *b1* or *b2* and a port name by *p1* or *p2*.

from the class itself).

3.4.4 Connecting stars and galaxies

The next step is to connect the blocks so that they can pass data among themselves using the `connect` command. This forms a connection between two stars (or galaxies) by connecting their portholes. A porthole is specified by giving the star (or galaxy) name followed by the port name within the star. The first porthole must be an output porthole and the second must be an input porthole. For example:

```
connect mystar output yourstar input
```

The connect command accepts an optional integer delay parameter. For example:

```
connect mystar output yourstar input 1
```

This specifies one delay on the connection. The delay parameter makes sense only for

command	arguments	description	page
`registerAction`	`pre | post` *command*	Register a Tcl command to be executed before or after stars fire.	3-15
`renameuniv`	*?oldname? newname*	Rename a universe (default: current universe).	3-2
`reset`	*?name?*	Empty a universe (default: “main”).	3-11
`run`	*?num?*	Run the current universe *num* times (default: 1).	3-10
`schedtime`	`?actual?`	Print the normalized (default) or unnormalized current scheduler time.	3-11
`schedule`		Generate and print a schedule (only valid for some domains).	3-10
`seed`	*number*	Change or print the random number seed.	3-13
`setstate`	*b1 state_name value*	Change the state of a block to *value*.	3-7
`source`	*filename*	Read commands from the specified file.	3-13
`star`	*name class*	Create a named instance of a star from the given class.	3-4
`stoptime`		Return the stop time of the current run.	3-10
`statevalue`	*b1 name* `?current | initial?`	Print the current or initial value of state *name* in block *b1*.	3-15
`target`	*?newtarget?*	Change or display the name of the current target.	3-12
`targetparam`	*name ?value?*	Change or display the value of a target state.	3-12
`targets`	*?domain?*	List targets usable with *domain* (default: current domain).	3-12
`topblocks`	*?block_or_classname?*	List top-level blocks of the named block (default: current galaxy).	3-15
`univlist`		List the names of all defined universes.	3-2
`wrapup`		Invoke the wrapup method of all the blocks.	3-10

TABLE 3-3: Final third of the summary of `ptcl` commands. Arguments are in *italic*; literals are in `Courier`; optional arguments are enclosed in question marks. A block name is indicated by *b1* or *b2* and a port name by *p1* or *p2*.

domains that support it. The delay argument may be an integer expression with variables referring to galaxy parameters as well.

One or both of the portholes may really be a `MultiPortHole`. If so, the effect of doing the connect is to create a new porthole within the `MultiPortHole` and connect to that (see also the `numports` command).

3.4.5 Netlist-style connections

As an alternative to issuing connect commands (which specify point-to-point connections) you may specify connections in a netlist style. This syntax is used to connect an output to more than one input, for example (this is called *auto-forking*). Two commands are provided for this purpose. The `node` command creates a node:

```
node nodename
```

The `nodeconnect` command connects a porthole to a node:

```
nodeconnect starname portname nodename ?delay?
```

Any number of portholes may be connected to a node, but only one of them can be an output node.

3.4.6 Bus connections between MultiPortHoles

A pair of multiportholes can be connected with a bus connection, which means that each multiporthole has *N* portholes and they all connect in parallel to the corresponding port in the other multiporthole. The syntax for creating such connections is

```
busconnect srcstar srcport dststar dstport width ?delay?
```

Here *`width`* is an expression specifying the width of the bus (how many portholes in the multiportholes); and *`delay`* is an optional expression giving the delay on each connection. The other arguments are identical to those of the `connect` command.

3.4.7 Connecting internal galaxy stars and galaxies to the outside

When you define a new galaxy there are typically external connections to that galaxy that need to be connected through to internal blocks. The `alias` command is used to add a porthole to the current galaxy, and associate it with an input or output porthole of one of the contained stars within the galaxy. An example is:

```
alias galaxyin mystar starin
```

This also works if *`starin`* is a `MultiPortHole` (the galaxy will then appear to have a multiporthole as well).

3.4.8 Defining parameters and states for a galaxy

A *state* is a piece of data that is assigned to a galaxy and can be used to affect its behavior. Typically the value of a state is coupled to the state of blocks within the galaxy, allowing you to customize the behavior of blocks within the galaxy. A *parameter* is the initial value of a state. The `newstate` command adds a state to the current galaxy. The form of the command is

```
newstate state-name state-class default-value
```

The *`state-name`* argument is the name to be given to the state. The *`state-class`* argument is the type of state. All standard types are supported (see table 2-6 on page 2-33). The *`default-value`* argument is the default value to be given to the state if the user of the galaxy does not change it (using the `setstate` command described below). The *`default-value`* specifies the initial value of the state, and can be an arbitrary expression involving constant values and other state names; this expression is evaluated when the simulation starts. The following state names are predefined: `YES`, `NO`, `TRUE`, `FALSE`, `PI`. `YES` and `TRUE` have value 1; `NO` and `FALSE` have value 0; `PI` has the value 3.14159... Some examples are:

```
        newstate count int 3
        newstate level float 1.0
        newstate title string "This is a title"
        newstate myfreq float galaxyfreq
        newstate angularFreq float "2*PI*freq"
```

The full syntax of state initial value strings depends on the type of state, and is explained in “Parameters and states” on page 2-14.

3.4.9 Setting the value of states

The `setstate` command is used to change the value of a state. It can be used in three contexts:

- Change the value of a state for a star within the current galaxy.
- Change the value of a state for a galaxy within the current galaxy.
- Change the value of a state within the current galaxy.

The latter would normally be used when you want to perform multiple simulations using different parameter values. The syntax for `setstate` is:

```
        setstate block-name state-name value
```

Here,

- *`block-name`* is either the name of a star or a galaxy that is inside the current galaxy, and it is the block for which the value of the state is to be changed. It can also be `this`, which says to change a state belonging to the current galaxy itself.
- *`state-name`* is the name of a state which you wish to change.
- *`value`* is the new value for the state. The syntax for *`value`* is the same as described in the `newstate` command. However, the expression for *`value`* may refer to the name of one or more states in the current galaxy or an ancestor of the current galaxy.

An example of the use of `setstate` is given in the section describing `defgalaxy` below.

3.4.10 Setting the number of ports to a star

Some stars in Ptolemy are defined with an unspecified number of multiple ports. The number of connections is defined by the user of the star rather than the star itself. The `numports` command applies to stars that contain such `MultiPortHoles`; it causes a specified number of `PortHoles` to be created within the `MultiPortHole`. The syntax is

```
        numports star portname n
```

where *`star`* is the name of a star within the current galaxy, *`portname`* is the name of a `Mul-`

`tiPortHole` in the star, and *`n`* is an integer, representing the number of `PortHole`s to be created. After the portholes are created, they may be referred to by appending `#`*`i`*, where *`i`* is an integer, to the multiporthole name, and enclosing the resulting name in quotes. The main reason for using this command is to allow the portholes to be connected in random order. Here is an example:

```
star summer Add
numports summer input 2
alias galInput summer "input#1"
connect foo output summer "input#2"
```

3.4.11 Defining new galaxies

The `defgalaxy` command allows the user to define a new class of galaxy. The syntax is

```
defgalaxy class-name {
      command
      command
      ...
}
```

Here *`class-name`* is the name of the galaxy type you are creating. While it is not required, we suggest that you have the name begin with a capital letter in accordance with our standard naming convention — class names begin with capital letters. The *`command`* lines may be any of the commands described above — `star`, `connect`, `busconnect`, `node`, `nodeconnect`, `numports`, `newstate`, `setstate`, or `alias`. The defined class is added to the known list, and you can then create instances of it and add them to other galaxies. An example is:

```
reset
domain SDF
defgalaxy SinGen {
      domain SDF
      # The frequency of the sine wave is a galaxy parameter
      newstate freq float "0.05"
      # Create a star instance of class "Ramp" named "ramp"
      star ramp Ramp
      # The ramp advances by 2*pi each sample
      setstate ramp step "6.283185307179586"
      # Multiply the ramp by a value, setting the frequency
      star gain Gain
      # The multiplier is set to "freq"
      setstate gain gain "freq"
      # Finally the sine generator
      star sin Sin
      connect ramp output gain input
      connect gain output sin input
      # The output of "sin" becomes the galaxy output
      alias output sin output
}
```

In this example, note the use of states to allow the frequency of the sine wave generator to be changed. For example, we could now run the sine generator, changing its frequency to “0.02”, with the interpreter input:

```
star generator SinGen
setstate generator freq "0.02"
star printer Printer
connect generator output printer input
run 100
```

You may include a `domain` command within a `defgalaxy` command. If the inside domain is different from the outside domain, this creates an object known as a `Wormhole`, which is an interface between two domains. An example of this appears in a later section.

3.5 Showing the current status

The following commands display information about the current state of the interpreter.

3.5.1 Displaying the known classes

The `knownlist` command returns a list of the classes of stars and galaxies on the known list that are usable in the current domain. The syntax is

```
knownlist
```

It is also possible to ask for a list of objects available in other domains; the command

```
knownlist DE
```

displays objects available in the `DE` (discrete event) domain.

3.5.2 Displaying information on a the current galaxy or other class

If invoked without an argument, the `print` command displays information on the current galaxy. If invoked with an argument, the argument is either the name of a star (or galaxy) contained in the current galaxy, or the name of a class on the known list, and information is shown about that star (or galaxy). The syntax is

```
print
print star-name
print star-class
```

The command

```
descriptor ?name?
```

will print a short description of a block in the current galaxy or on the known list, or of the current galaxy if *name* is omitted. The commands

```
listobjs states ?name?
listobjs ports ?name?
listobjs multiports ?name?
```

will list the names of the states, ports, or multiportholes associated with the named star or galaxy.

3.6 Running the simulation

Once a simulation has been constructed using the commands previously described (also see the `source` command in “Loading commands from a file” on page 3-13), use the commands in this section to run the simulation.

3.6.1 Creating a schedule

The `schedule` command generates and returns the schedule (the order in which stars are invoked). For domains such as DE, this command returns a not-implemented message (since there is no “compile time” DE schedule as there is for SDF). The syntax is:

```
schedule
```

3.6.2 Running the simulation

The `run` command generates the schedule and runs it *n* times, where *n* is the argument (the argument may be omitted; its default value is 1). For the DE interpreter, this command runs the simulation for *n* time units, and *n* may be a floating point number (default 1.0). If this command is repeated, the simulation is started from the beginning. If animation is enabled, the full name of each star will be printed to the standard output when the star fires. The syntax is:

```
run
run n
```

3.6.3 Continuing a simulation

The `cont` command continues the simulation for *n* additional steps, or time units. If the argument is omitted, the default value of the argument is the value of the last argument given to a `run` or `cont` command (1.0 if no argument was ever given). The syntax is

```
cont
cont n
```

3.6.4 Wrapping up a simulation

The `wrapup` command calls the wrapup method of the current target (which, as a rule, will call the `wrapup` method of each star), signaling the end of the simulation run. The syntax is

```
wrapup
```

3.6.5 Interrupting a simulation

The command

```
halt
```

requests a halt of the currently executing simulation. Note that the halt does not occur immediately. This merely registers the request with the scheduler. This is especially useful within Tcl stars.

3.6.6 Obtaining the stop time of the current run

The command

```
stoptime
```

returns the time until which the current simulation will run. Tcl/Tk stars can use this command in their setup or go methods to find out the stop time of the current run.

3.6.7 Obtaining time information from the scheduler

The command

```
schedtime
```

returns the current time from the top-level scheduler of the current universe. If the target has a parameter named “schedulePeriod”, then the returned time is divided by this value. The command

```
schedtime actual
```

returns the scheduler time without dividing by “schedulePeriod.”

In SDF, `schedtime actual` should return the number of iterations. In SDF, “schedulePeriod” is usually set to 0, since in SDF has no notion of time, and to a timed domain, such as DE, SDF universes appear to fire instantaneously.

3.6.8 Animating a simulation

The `animation` command can be used to display on the standard output the name of each star as it runs. The syntax

```
animation on
```

enables animation, while

```
animation off
```

disables it. The syntax

```
animation
```

simply tells you whether animation is enabled or disabled.

3.7 Undoing what you have done

The commands in this section remove part or all of the structure you have built with previous commands.

3.7.1 Resetting the interpreter

The `reset` command replaces the universe `main` or a named universe by an empty universe. Any `defgalaxy` definitions you have made are still remembered. The syntax is

```
reset
reset universe_name
```

3.7.2 Removing a star

The `delstar` command removes the named star from the current galaxy. The syntax is

```
delstar name
```

where *name* is the name of the star.

3.7.3 Removing a connection

The `disconnect` command reverses the effect of a previous `connect` or `nodeconnect` command. The syntax is

```
        disconnect starname portname
```

where `starname` and `portname`, taken together, specify one of the two connected portholes. Note that you can disconnect by specifying either end of a porthole for a point-to-point connection.

3.7.4 Removing a node

The `delnode` command removes a node from the current galaxy. Syntax:

```
        delnode node
```

3.8 Targets

Ptolemy uses a structure called a *target* to control the execution of a simulation, or, in code generation, to control code generation, compilation, and execution. There is always a target; by default (if you issue no target commands), your target will have the name `default-XXX`, where `XXX` is replaced by the name of the current domain. Alternative targets for simulation can be used to specify different behavior (for example, to use a different scheduler or to analyze a schematic rather than running a simulation). For code generation, the target contains information about the target of compilation, and has methods for downloading code and starting execution.

3.8.1 What targets are available?

The command

```
        targets
```

returns the list of targets available for the current domain. The command

```
        targets domain
```

returns the list of targets available for *domain*.

3.8.2 Changing the target

The command

```
        target
```

displays the target for the current universe or current galaxy, together with its parameters. Specifying an argument, e.g.

```
        target new-target-name
```

changes the target to `new-target-name`.

3.8.3 Changing target parameters

Target parameters may be queried or changed with the `targetparam` command. The syntax is

```
        targetparam param-name ?new-value?
```

3.8.4 Pragmas

Ptolemy can use target pragmas as a generalization of the attribute mechanism to inform the target of the user's wishes. The Dynamic Dataflow (DDF) domain uses pragmas to specify the number of firings of a star required in one iteration. The C Code Generation (CGC) domain uses pragmas to identify any parameters that the user would like to change on the command line. See “Setting Parameters Using Command-line Arguments” on page 14-4.

`pragma` *`b1 b2 name value`*

Set pragma *`name`* to *`value`* for block *`b2`* in parent *`b1`*.

`pragmaDefaults` *`target`*

Print the default values of the pragmas for the target.

3.9 Miscellaneous commands

This section describes the remaining interpreter commands.

3.9.1 Loading commands from a file

For complicated simulations it is best to store your interpreter commands—at least those defining the simulation connectivity—in a file rather than typing them into the interpreter directly. This way you can run your favorite editor in one window and run the interpreter from another window, easily modifying the simulation and also keeping a permanent record. Two exceptions to this are changing states using the `setstate` command and running and continuing the simulation using `run` and `cont`—this is normally done interactively with the interpreter.

The `source` command reads interpreter commands from the named file, until the end of the file or a syntax error occurs. The “#” character indicates that the rest of the line is a comment. By convention, files meant to be read by the load command end in “.pt”. Example:

```
source "testfile.pt"
```

The tilde notation for users’ home directories is allowed; for example, if your installation of Ptolemy was made by creating a user `ptolemy` (see “Setup” on page 2-1), try

```
source "$PTOLEMY/demo/ptcl/sdf/basic/butterfly.pt"
```

It is also possible to specify a file to be loaded by the interpreter on the command line. If, when you start the interpreter you type

```
ptcl myCommands.pt
```

the interpreter will load the named file, execute its commands, and then quit. No command prompt will appear. The `source` command is actually built into Tcl itself, but it is described here nevertheless, for convenience.

3.9.2 Changing the seed of random number generation

The `seed` command changes the seed of the random number generation. The default value is 1. The syntax is

`seed` *`n`*

where *`n`* is an unsigned integer.

3.9.3 Changing the current directory

The `cd` command changes the current directory. For example,

```
cd "$PTOLEMY/demo/ptcl/sdf/basic"
source "butterfly.pt"
```

will load the same file as the example in the previous section. Again, we have assumed that your installation contains a user `ptolemy` (see “Setup” on page 2-1). To see what the interpreter’s current directory is, you can type

```
pwd
```

3.9.4 Dynamically linking new stars

The interpreter has the ability to extend itself by linking in outside object files; the object files in question must define single stars (they will have the right format if they are produced from preprocessor input). Unlike `pigi`, the graphical interface, the interpreter will not automatically run the preprocessor and compiler; it expects to be given object files that have already been compiled. The syntax is

```
link object-file-name
```

Any star object files that are linked in this way must only call routines that are already statically or permanently linked into the interpreter. For that reason, it is possible that a star that can be linked into `pigi` might not be linkable into the interpreter, although this is rare. Specifically, `pigi` contains Tk, an X window toolkit based on Tcl, while `ptcl` does not. Hence, any star that uses Tk is excluded from `ptcl`.

Building object files for linking into Ptolemy can be tricky since the command line arguments to produce the object file depend on the operating system, the compiler and whether or not shared libraries are used. `$PTOLEMY/mk/userstars.mk` includes rules to build the proper object file for a star. See “Dynamic linking fails” on page A-30. for hints about fixing incremental linking problems.

It is also possible to link in several object files at once, or pull in functions from libraries by use of the `multilink` command. The syntax is

```
multilink opt1 opt2 opt3 ...
```

where the options may be the names of object files, linker options such as “-L” or “-l” switches, etc. These arguments are supplied to the Unix linker along with whatever options are needed to completely specify the incremental link.

When the above linker commands are used, the linked code has temporary status; symbols for it are not entered into the symbol table (meaning that the code cannot be linked against by future incremental links), and it can be replaced; for example, an error in the loaded modules could be corrected and the `link` or `multilink` command could be repeated. There is an alternative linking command that specifies that the new code is to be considered “permanent”; it causes a new symbol table to be produced for use in future links (See the ptlang `derivedFrom` item in the Ptolemy Programmers Manual for more information). Such code cannot be replaced, but it can be linked against by future incremental link commands. The syntax is

```
permlink opt1 opt2 opt3 ...
```

where the options are the same as for the `multilink` command.

3.9.5 Top-level blocks

The command

```
topblocks
```

returns the list of top-level blocks in the current galaxy or universe. With an argument,

```
topblocks block
```

it returns the list of top-level blocks in the named block.

3.9.6 Examining states

The `statevalue` command takes the form

```
statevalue block state
```

and returns the current value of the state *`state`* within the block *`block`*. The command takes an optional third argument, which may be either `“current”` to specify that the current value should be returned (the default), or `“initial”` to specify that the initial value (the parameter value) should be returned.

3.9.7 Giving up

The `exit` command exits the interpreter. The syntax is

```
exit
```

3.9.8 Getting help

The `help` command implements a simple help system describing the commands available and their syntax. It does not provide help with the standard Tcl functions. The syntax is

```
help topic
```

or

```
help ?
```

for a list of topics. If the argument is omitted, a short "help on help" is printed.

3.9.9 Registering actions

It is possible to associate a Tcl action with the firing of any star. The `registerAction` command does this. The syntax is

```
registerAction pre tcl_command
registerAction post tcl_command
```

The first argument specifies whether the action should occur before or after the firing of a star. The second argument is a string giving the first part of a tcl command. Before this command is invoked, the name of the star that triggered the action will be appended as an argument. For example:

```
registerAction pre puts
```

will result in the name of a star being printed on the standard output before it is fired. A typical “action” resulting from this command would be

```
puts universe_name.galaxy_name.star_name
```

The value returned by `registerAction` is an “action_handle”, which must be used to cancel the action using `cancelAction`. The syntax is

```
set action_handle [registerAction pre tcl_command]
cancelAction action_handle
```

3.9.10 The Interface to Matlab and Mathematica

Ptcl can control Matlab [Han96] and Mathematica [Wol92] processes by means of the `matlab` and `mathematica` commands. The commands have a similar syntax:

```
matlab command ?arg1? ?arg2?
mathematica command ?arg1? ?arg2?
```

The `matlab` command controls the interaction with a shared Matlab process. The possible commands and arguments are:

command	arguments	description
`end`		terminate a session with Matlab
`eval`	*script*	evaluate a Matlab script and print the result
`get`	*name script*	evaluate a Matlab script and get the named Matlab matrix as Tcl lists of numbers
`getpairs`	*name script*	evaluate a Matlab script and get the named Matlab matrix as ordered pairs of numbers
`send`	*script*	evaluate a Matlab script and suppress the output
`set`	*name rows cols real imag*	set the named Matlab matrix with real and imaginary values
`start`		start a new Matlab session
`status`		return the status of the Tcl/Matlab connection (0 means connected, -1 means not initialized, and 1 means error)
`unset`	*name*	unset the named Matlab matrix

The `mathematica` command controls the interaction with a shared Mathematica process. The possible commands and arguments are

command	arguments	description
`end`		terminate a session with Mathematica
`eval`	*script*	evaluate a Mathematica script and print the result
`get`	*name script*	evaluate a Mathematica script and get the named Mathematica variable as a Tcl string
`send`	*script*	evaluate a Mathematica script and suppress the output
`start`		start a new Mathematica session
`status`		return the status of the Tcl/Mathematica connection (0 means connected, -1 means not initialized, and 1 means error)

To initiate a connection to a Matlab and Mathematica process, use

```
matlab start
mathematica start
```

To generate a simple plot of a straight line in Matlab and Mathematica, use

```
matlab send { plot([0 1 2 3])}
mathematica send { Plot[x, {x, 0, 3} ] }
```

The `send` command suppresses the output normally returned by interacting with the program using the command interface. The `eval` command, on the other hand, returns the dialog with the console interface:

```
        mathematica eval { Plot[x, {x, 0, 3}] }
        -Graphics-
```

To terminate the connection, use

```
        matlab end
        mathematica end
```

One can work with matrices as Tcl lists or in Matlab format. To create a new Matlab matrix x that has two rows and three columns:

```
        matlab set x 2 3 "1 2 3 4 5 6" "1 1 1 1 1 1"
```

We can retrieve this Matlab matrix in the same format:

```
        matlab get x
        2 3 {1.0 2.0 3.0 4.0 5.0 6.0} {1.0 1.0 1.0 1.0 1.0 1.0}
```

We can also retrieve the matrix elements as a Tcl list of complex numbers in an ordered-pair format:

```
        matlab getpairs x
        (1.0,1.0) (2.0,1.0) (3.0,1.0) (4.0,1.0) (5.0,1.0) (6.0,1.0)
```

Now, matrices can be manipulated in both Tcl and Matlab.

Javier Contreras contributed the following example that creates a Tcl list, sends it to MATLAB as a 2x2 matrix, calculates the inverse in MATLAB and retrieves it back to Tcl as list and/or pairs.

```
      ptcl> matlab start
      ptcl> set a 1
      1
      ptcl> set b 2
      2
      ptcl> set c 3
      3
      ptcl> set d 4
      4
      ptcl> set e [expr "{$a $b $c $d}"]
      1 2 3 4
      ptcl> set f [expr "{$a $b $c $d}"]
      1 2 3 4
      ptcl> matlab set matrix $b $b $e $f
      ptcl> matlab eval {matrix(1,1)}
      >>
      ans =

      1.0000 + 1.0000i


      ptcl> set inv_matrix [matlab get inverse {inverse = inv(matrix)}]
      2 2 {-1.0 0.5 0.75 -0.25} {1.0 -0.5 -0.75 0.25}
      ptcl> set inv_matrix [matlab getpairs inverse {inverse =
      inv(matrix)}]
      (-1.0,1.0) (0.5,-0.5) (0.75,-0.75) (-0.25,0.25)
      ptcl> set new $inv_matrix
```

```
(-1.0,1.0) (0.5,-0.5) (0.75,-0.75) (-0.25,0.25)
ptcl> lindex $new 0
(-1.0,1.0)
ptcl> matlab unset matrix
ptcl> matlab eval {matrix(1,1)}
ptcl> matlab end
```

For other examples of the use of the matlab and mathematica Ptcl commands, see “Using Matlab and Mathematica to Compute Parameters” on page 2-18. These commands support the Matlab and Mathematica consoles in Tycho.

3.10 Limitations of the interpreter

There should be many more commands returning information on the simulation, to permit better exploitation of the full power of the Tcl language.

3.11 A wormhole example

Here is an example of a simulation that contains both an SDF portion and a DE portion. In this example, a Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples. This demo shows how easy it is to use the SDF stars to perform computation on DE particles. The overall delay of the SDF wormhole is zero, so the result is simply Poisson arrivals of Gaussian noise samples.

A `Wormhole` has an *outer* domain and an *inner* domain. The outer domain is determined by the current domain at the time the user starts the `defgalaxy` command to create the wormhole. The inner domain is determined by the `domain` command that appears inside the galaxy definition.

```
reset
# create the wormhole
domain DE
defgalaxy wormBody {
      domain SDF
      star add Add; numports add input 2
      star IIDGaussian1 IIDGaussian
      alias out add output
      alias in add "input#1"
      connect IIDGaussian1 output add "input#2"
}
# Creating the main universe.
domain DE
star wormBody1 wormBody
star Poisson1 Poisson; star graf XMgraph
numports graf input 2
setstate graf title "Noisy Poisson Process"
setstate graf options "-P -0 original -1 noisy"
node node1
nodeconnect Poisson1 output node1
nodeconnect wormBody1 in node1
nodeconnect graf "input#1" node1
```

```
connect wormBody1 out graf "input#2"
run 40
wrapup
```

3.12 Some hints on advanced uses of ptcl with pigi

Although we have not had time to pursue it aggressively in this release, flexible control of Ptolemy simulations (e.g. executing a simulation many times with different parameter settings) is now possible. This can be done by using `ptcl` and `pigi` together.

Warning: This mechanism is still under development, so please note that what is described in this section is likely to change.

3.12.1 Ptcl as a simulation control language for pigi

If you start `pigi` with the `-console` option, then a console window will appear that will accept `ptcl` commands. To experiment with this, open the `sinMod` demo in the SDF basic demo palette, and execute the `pigi` command `compile-facet` (in the `Exec` submenu). This command reads the oct facet from disk, and constructs the Ptolemy data structures to represent it in memory. In your console window, you should see the prompt:

```
pigi>
```

Note what happens if you ask for the name of the current universe:

```
pigi> curuniverse
sinMod
pigi>
```

By compiling the facet, you have created a universe called `sinMod`, and made it the current universe. If you just started `pigi`, then this is one of only two universes in existence:

```
pigi> univlist
main sinMod
pigi>
```

The universe `main` is the default, empty universe that Ptolemy starts with. To verify the contents of the `sinMod` universe, use the `print` command:

```
pigi> print
GALAXY: sinMod
Descriptor: An interpreted galaxy
Contained blocks: singen2 modulator1 XMgraph.input=11
pigi>
```

You can execute this universe from the console window:

```
pigi> run 400
pigi> wrapup
```

Notice that you will not see any output until you invoke the wrapup command, since the `XMgraph` star creates the output plot in its wrapup method.

So far, you have not done anything you could not have done more directly using `pigi`. However, you can change the value of parameters from `ptcl`. To do this, you must first determine the name of the instance of the star or galaxy with the parameter you want to control. Place the mouse over the `singen` icon in the `sinMod` galaxy, and issue the pigi `show-name` (‘n’) command. Most likely, the name will be `singen2`, although it could be different on successive runs. This is an instance name generated automatically by `pigi`. Notice that it is the name shown by the print command above. Also, use the `edit-params` (‘e’) command over the `singen` icon to determine that `singen2` has a parameter named `frequency` with value `PI/100`. Now try the following commands:

```
pigi> setstate singen2 frequency PI/50
pigi> run 400
pigi> wrapup
```

Notice that the frequency of the modulating sinusoid is now twice as high as before.

Much more interestingly, you can now construct a series of runs using Tcl as a scripting language:

```
pigi> foreach i {0.25 0.5 0.75 1 1.25 1.5} {
pigi? setstate singen2 frequency $i*PI/100
pigi? setstate XMgraph.input=11 title \
pigi? "message frequency = [expr 0.01*$i]*PI"
pigi? run 400
pigi? wrapup
pigi? }
pigi>
```

This will invoke six runs, each with a different frequency parameter for the `singen` galaxy `singen2`. The `foreach` command is a standard Tcl command. Notice that in the third and fourth lines, we have also set the title parameter of the `XMgraph` star. This is advisable because otherwise it might be very difficult to tell which result corresponded to which run. Notice that the name of the `XMgraph` instance is “`XMgraph.input=11`”. It is a more complicated name because the icon is specialized to have only a single input port.

Using the full power of the Tcl language, the above mechanism can become extremely powerful. To use its full power, however, you will most likely want to construct your Tcl scripts in files. These files can even include the universe definition, as explained below, so you can create scripts that can be run under `ptcl` only, independent of `pigi`.

3.12.2 The pigi log file pigiLog.pt

In each `pigi` session, a log file named `pigiLog.pt` is generated in the user’s home directory. Every time an oct facet that represents a Ptolemy galaxy or universe is compiled, for example when running a simulation, the equivalent `ptcl` commands building the galaxy or universe are logged in pigiLog.pt. For example, if you followed the above procedure, opening the `sinMod` demo and issuing the `compile-facet` command, your `pigiLog.pt` file will contain something like the following:

```
reset
domain SDF
defgalaxy singen {
     domain SDF
     newstate sample_rate FLOAT "2*PI"
     newstate frequency FLOAT "PI/50"
     newstate phase_in_radians float 0.0
     star Ramp1 Ramp
```

```
          setstate Ramp1 step "2*PI*frequency/sample_rate"
          setstate Ramp1 value phase_in_radians
          star Sin1 Sin
          connect Ramp1 output Sin1 input
          alias out Sin1 output
     }
     defgalaxy modulator {
          domain SDF
          newstate freq FLOAT 0.062832
          star "Mpy.input=21" Mpy
          numports "Mpy.input=21" input 2
          star singen1 singen
          setstate singen1 sample_rate "2*PI"
          setstate singen1 frequency freq
          setstate singen1 phase_in_radians 0.0
          alias in "Mpy.input=21" "input#1"
          alias out "Mpy.input=21" output
          connect singen1 out "Mpy.input=21" "input#2"
     }
     newuniverse sinMod SDF
     target default-SDF
          targetparam logFile ""
          targetparam loopScheduler NO
          targetparam schedulePeriod 10000.0
          star singen2 singen
          setstate singen2 sample_rate "2*PI"
          setstate singen2 frequency "PI/100"
          setstate singen2 phase_in_radians 0.0
          star modulator1 modulator
          setstate modulator freq "0.2*PI"
          star "XMgraph.input=11" XMgraph
          numports "XMgraph.input=11" input 1
          setstate "XMgraph.input=11" title "A modulator demo"
          setstate "XMgraph.input=11" saveFile ""
          setstate "XMgraph.input=11" options "=800x400+0+0 -0 x"
          setstate "XMgraph.input=11" ignore 0
          setstate "XMgraph.input=11" xUnits 1.0
          setstate "XMgraph.input=11" xInit 0.0
          connect singen2 out modulator1 in
          connect modulator1 out "XMgraph.input=11" "input#1"
```

This is a `ptcl` definition of a universe that is equivalent to the oct facet. In normal usage, you may need to edit this file considerably to extract the portions you need, because all the galaxies and universes compiled in a `pigi` session are logged in the same log file. Also, as of this writing, the file does not necessarily get flushed after your compile-facet command completes, so the last few lines may not appear until more lines are written to the file, or you exit `pigi`.

Note that `pigi` compiles the sub-galaxies recursively before compiling the top-level universe. Therefore, the `ptcl` definitions are generated and logged in this recursive order. For instance, in the pigiLog.pt shown above, `ptcl` definitions of the `singen` and `modulator` galaxies appear before that of the `sinMod` universe. Also, if a galaxy has been compiled before, and thus is on the knownlist, its `ptcl` definition will not be generated and logged again when it is used in another universe.

One use of the `ptcl` definitions obtained from pigiLog.pt is to submit bug reports. It is the best way to describe in ASCII text the Ptolemy universe that causes problems.

3.12.3 Using pigiLog.pt to build scripts

If you restart `pigi`, run the `sinMod` demo in the SDF basic demo palette once, then quit `pigi`, then your `pigiLog.pt` file will be as above. Make a copy of `pigiLog.pt` and name it, say, `sinMod.pl`.

To run this simulation with different message waveform frequencies, you may do the following in `ptcl`, analogous to the above commands in `pigi`:

```
# build the sinMod universe
source sinMod.pl
foreach i {0.25 0.5 0.75 1 1.25 1.5} {
     # set parameter values
     setstate singen2 frequency $i*PI/100
     setstate XMgraph.input=11 title \
     "message frequency = [expr 0.01*$i]*PI"
     # execute it
     run 400
     wrapup
}
```

The combination of `ptcl` and `pigi` is very powerful. The above are just some hints on how they can be used together.

3.12.4 oct2ptcl

Kennard White’s program `oct2ptcl` can be used to convert Ptolemy facets to ptcl code. `Oct2ptcl` is not part of the default distribution, and it is not built automatically. You can find the `oct2ptcl` sources in the other.src tar file in `ptolemy/src/octtools/tkoct/oct2ptcl`. `oct2ptcl` is not formally part of Ptolemy, but some developers may find it useful.

Chapter 4. Introduction to Domains, Targets, and Foreign Tool Interfaces

Authors: *Joseph T. Buck*
Brian L. Evans
Soonhoi Ha
Asawaree Kalavade
Edward A. Lee
Thomas M. Parks

Michael C. Williamson

Other Contributors: *The entire Ptolemy team*

4.1 Introduction

The Ptolemy software architecture is described in Chapter 1 and shown in Figure 1-2. The Ptolemy kernel provides a basic set of C++ classes and functions used by the rest of Ptolemy, but it does not implement any particular model of computation. Instead, a model of computation is defined by a domain. A domain defines the semantics of the model, but not how the computations are performed. The computations can be performed using one or more implementation languages, such as C++, C, MATLAB, and VHDL. A target coordinates the scheduling and implementation of algorithms described in a particular domain. As part of the coordination, a target may provide an interface to software (compiler, assembler, simulator, etc.) or hardware. A typical domain supports many different types of schedulers and many different implementation technologies, which is possible by having many different targets. Over twenty domains have been developed for Ptolemy, and 14 are released in Ptolemy 0.7; of these, nine support multiple targets.

In Ptolemy, a complex system is specified as a hierarchical composition (nested tree structure) of simpler subsystems. Each subsystem is modeled by a domain. A subsystem (also called a galaxy) can internally use a different domain than that of its parent or children. In mixing domains, the key is to ensure that at the interface, the child galaxy obeys the semantics of the parent domain. This interface is called a wormhole. Ptolemy does not yet make the wormhole mechanism foolproof. Any domain can be used at the top level.

As shown in Figure 1-2, Ptolemy consists of dataflow, discrete-event, and control-oriented families of domains. The Synchronous Dataflow (SDF) and Discrete-Event (DE) domains are the most mature. In terms of semantics, the Discrete-Event domains are the furthest from the dataflow domains in the 0.7 distribution. Other domains with semantics very different from dataflow, are the Finite State Machine (FSM) and Synchronous/Reactive (SR) domains.

Domains perform either simulation or code generation. Simulation domains are interpreters that run an executable specification of a system on the local workstation. Code generation domains translate the specification into some language such as C or VHDL and then optionally manage the execution of that generated code. In 0.6 and later, code generation domains can be mixed with each other and with simulation domains. Thanks to José Pino for developing hierarchical scheduling to support this capability.

The model of computation is the *semantics* of the network of blocks. It defines what is meant by an interconnection of blocks, and how the interconnection will behave when executed. The domain determines the model of computation, but in the case of code generation domains, it also determines the target language. So for example, the CGC (Code Generation in C), C50 (Code Generation for the Texas Instruments TMS320C50) and the CG56 (Code Generation for the Motorola DSP56000) domains all use the synchronous dataflow model of computation (the same as the SDF domain). The CGC domain also supports features of the Boolean dataflow (BDF) domain, which supports a measure of run-time scheduling in a very efficient way.

Simulation domains can be either timed or untimed. Untimed domains carry no notion of time in their semantic model. Instead of chronology, they deal only with the order of particles or actions. Timed domains have a notion of *simulated time*, where each particle or action is modeled as occurring at some particular point in this simulated time. Particles and actions are processed chronologically. Timed and untimed domains can be mixed. From the perspective of a timed domain, actions in an untimed domain will appear to be instantaneous. Moreover, timed domains can exist at several levels of the hierarchy, or in parallel at a given level of the hierarchy, separated by untimed domains, and their chronologies will be synchronized. That is, the notion of simulated time in Ptolemy is a global notion. When particles and actions are processed chronologically in each timed domain that is present, then they will be processed chronologically globally.

In this chapter, we also introduce the `Target` class. The basic role of this class is in managing design flow. In a simulation domain, the target selects the scheduler to use (there can be several schedulers in a single domain) and starts and stops a simulation. In a code generation domain, the target also selects the scheduler, but then also generates the code, compiles it, and runs it on a suitable platform. Targets can be defined hierarchically; for example, a multiprocessor target may consist of several, possibly heterogeneous execution platforms, each specified itself as a target. In this example, the top level target might handle the partitioning and interprocessor communication, and the lower level targets might handle the code generation, compilation, and execution. Targets play a much bigger role in code generation domains than in simulation domains.

Ptolemy users often prematurely set out to make a new domain. While it is the intent of Ptolemy to support such experimentation, this task should be undertaken with some trepidation. Although any software engineer can create a domain that will work, defining a useful and correct model of computation is a much harder task. It is very easy, for example, to define a non-determinate model of computation. This means that the behavior of an application will depend on implementation details in the scheduler that are not explicitly known to the user. As a consequence, a user make a small, seemingly innocuous change in an application, and unexpectedly get radically different behavior. At Berkeley, many more domains have been built than are currently distributed. Sometimes, domains have been discarded because of unexpected subtleties in the model of computation. In other cases, domains have been built on top of third-party software or hardware that has become obsolete.

A prerequisite for creating any new domain is understanding the existing domains in Ptolemy. Frequently, one of these domains will meet your needs with simpler extensions, like a new target or a family of stars. If, for example, you are unhappy with the performance of a scheduler, it may make more sense to define a new scheduler (and a target to support it) within an existing domain, rather than creating a new domain.

This chapter gives a brief introduction to the simulation and code generation domains released in Ptolemy 0.7. It also highlights the domains that were present in earlier versions of Ptolemy but are no longer released. This chapter ends with an overview of the interfaces to foreign tools, such as simulators, interpreters, and compilers.

4.2 Synchronous dataflow (SDF)

The SDF domain in Ptolemy is the oldest, most mature domain. Much of its basic capability was ported from Gabriel, the predecessor system [Bie90][Lee89], although it has been extended considerably. SDF is a special case of the dataflow model of computation developed by Dennis [Den75]. The specialization of the model of computation is to those dataflow graphs where the flow of control is completely predictable at compile time. It is a good match for synchronous signal processing systems, those with sample rates that are rational multiples of one another.

The SDF domain is suitable for fixed and adaptive digital filtering, in the time or frequency domains. It naturally supports multirate applications, and its rich star library includes polyphase real and complex FIR filters. Applications with examples in the demo library include speech coding, sample-rate conversion, analysis-synthesis filter banks, modems, phase-locked loops, channel simulation, linear prediction, chaos, filter design, Kalman filtering, phased array beamforming, spectral estimation, sound synthesis, image processing, and video coding. The SDF domain has been used for a number of years at Berkeley for instruction in signal processing, at both the graduate and undergraduate level. The exercises that are assigned to the students are included in the SDF chapter.

4.3 Higher-Order Functions (HOF)

A function is *higher-order* if it takes a function as an argument and/or returns a function. A classic example is *mapcar* in Lisp, which takes two arguments, a function and a list. Its behavior is to apply the function to each element of the list and to return a list of the results. The HOF domain implements a similar function, in the form of a star called `Map`, that can apply any other star (or galaxy) to the sequence(s) at its inputs. Many other useful higher-order functions are also provided by this domain.

The HOF domain provides a collection of stars designed to be usable in all other Ptolemy domains. It is intended to be included as a subdomain by all other domains.

4.4 Dynamic dataflow (DDF)

The predictable control flow of SDF allows for efficient scheduling, but limits the range of applications. In particular, data-dependent flow of control is only allowed within the confines of a star. To support broader applications, the DDF domain uses dynamic (run-time) scheduling. For long runs, involving many iterations, this is more expensive than the static scheduling that is possible with SDF. But in exchange for this additional cost, we get a model of computation that is as versatile as that of conventional programming languages. It supports conditionals, data-dependent iteration, and true recursion.

Although the DDF domain is, in principle, a fully general programming environment, it is nonetheless better suited to some applications than others. We have found that signal processing applications with a limited amount of run-time control are a good match. Examples include systems with multiple modes of operation, such as modems (which have start-up sequences and often implement multiple standards), signal coding algorithms (which often offer a range of compression schemes), and asynchronous signal processing applications, such as timing recovery and arbitrary sample-rate conversion. The demos provided with the domain show how to realize conditionals, iteration, and recursion.

The SDF domain is a subdomain of DDF, which means that SDF stars can be used in DDF systems. For greater efficiency on long runs, the two domains can also be mixed using the Ptolemy hierarchy. A galaxy within a DDF system can be SDF, meaning that it will use an SDF scheduler. Conversely, a galaxy within an SDF system can be DDF.

4.5 Boolean dataflow (BDF)

Boolean dataflow was developed by Joe Buck as part of his Ph.D. thesis research [Buc93c]. Like DDF, it supports run-time flow of control. Unlike DDF, it attempts to construct a compile-time schedule. Thus it achieves the efficiency of SDF with the generality of DDF. It currently supports a somewhat more limited range of stars than DDF, and does not support recursion, but the model of computation is, in principle, equally general. Its applications are the same as those of DDF.

The basic mechanism used in BDF is to construct an *annotated schedule*, by which we mean a static schedule where each firing in the schedule is annotated with the Boolean conditions under which it occurs. Thus, any sequence of firings can depend on a sequence of Boolean values computed during the execution. Executing the annotated schedule involves much less overhead than executing a dynamic dataflow schedule.

4.6 Process Network (PN)

The process network domain, created by Thomas M. Parks and documented in his Ph.D. thesis [Par95], implements Kahn process networks, a generalization of dataflow where processes replace actors. It has some of the flavor of the recently removed CP domain, in that it implements concurrent processes, but unlike the CP domain, it is determinate and has no model of time. The PN domain is implemented using POSIX threads. In principle, PN systems can run in parallel on multiprocessor workstations with appropriate OS support for threads.

The SDF, BDF and DDF domains are subdomains of PN, which means that these stars can be used directly in PN systems. When stars from these domains are used in a PN system, each dataflow actor becomes a dataflow process [Lee95]. For greater efficiency, dataflow domains can be mixed with PN using the Ptolemy hierarchy. A galaxy within a PN system can be SDF, BDF, or DDF, using a scheduler appropriate for that domain. The galaxy as a whole becomes a single process in the PN system.

4.7 Synchronous Reactive (SR)

The Synchronous Reactive domain, created by Stephen Edwards and documented in his Ph.D. thesis [Edw97], is a new and very experimental domain. The Synchronous Reactive domain is a statically-scheduled simulation domain in Ptolemy designed for concurrent, control-dominated systems. To allow precise control over timing, it adopts the synchronous model of time, which is logically equivalent to assuming that computation is instantaneous

SR is similar to existing Ptolemy domains, but differs from them in important ways. Like Synchronous Dataflow (SDF), it is statically scheduled and deterministic, but it does not have buffered communication or multi-rate behavior. SR is better for control-dominated systems that need control over when things happen relative to each other; SDF is better for data-dominated systems, especially those with multi-rate behavior.

SR also resembles the Discrete Event (DE) domain. Like DE, its communication channels transmit events, but unlike DE, it is deterministic, statically scheduled, and allows zero-delay feedback loops. DE is better for *modeling* the behavior of systems (i.e., to better understand their behavior), whereas SR is better for *specifying* a system's behavior (i.e., as a way to actually build it).

4.8 Finite State Machine (FSM)

The Finite State Machine domain, created by Bilung Lee, is a new and very experimental domain. The finite state machine (FSM) has been one of the most popular models for describing control-oriented systems, e.g., real-time process controllers. The FSM domain uses Tycho graphical user interface for the specification of FSM blocks. Currently FSM can interoperate with SDF and DE domains.

4.9 Discrete Event (DE)

The DE domain is a relatively mature domain using an event-driven model of computation. In this domain, particles carry time stamps, and represent events that occur at arbitrary points in simulated time. Events are processed in chronological order. Two schedulers are available. The default scheduler is based on the "calendar queue" mechanism developed by Randy Brown and was written by Anindo Banerjea and Ed Knightly. Since this scheduler is relatively new, the older and simpler but less efficient scheduler is also provided.

DE schedulers maintain an event queue, which is a list of events sorted chronologically by time stamp. The scheduler selects the next event on the list, and determines which star should be fired to process the event. The difference between the efficient calendar queue scheduler and the naive simple scheduler is in the efficiency with which this queue is updated and accessed. Considerable effort was put into consistent and predictable handling of simultaneous events.

The DE domain is suitable for high-level modeling of communications networks, queueing systems, hardware systems, and transportation networks. The demos included with the domain include a variety of queueing systems, shared resource management, communication network protocols, packet-switched networks, wireless networks, and multimedia systems. The latter class of applications take advantage of the ability that Ptolemy has to mix domains by modeling speech and video encoding algorithms using the SDF domain and a packet switched network using the DE domain. There are also some more specialized uses of the DE domain, such as modeling shot noise and synchronizing a simulation to a real-time clock.

4.10 Multidimensional Synchronous Dataflow (MDSDF)

The MDSDF domain was developed by Mike Chen and is still very experimental. This domain is an extension of the Synchronous Dataflow model to multidimensional streams and is based on the work of Edward Lee [Lee93b]. MDSDF provides the ability to express a greater variety of dataflow schedules in a graphically compact way. It also allows nested resettable loops and delays. Additionally, MDSDF has the potential for revealing data parallelism in algorithms. The current implementation of the MDSDF domain only allows two dimensional streams, although we hope that many of the ideas used in the development of the domain can be generalized to higher dimensions.

For a full discussion of the MDSDF domain in Ptolemy, see [Che94].

4.11 Code generation (CG)

The CG domain is the base from which all other code generation domains (such as CGC and CG56) are derived. This domain supports a general dataflow model equivalent to the BDF and SDF models. The stars in this domain do little more than generate comments when fired, but they can serve to demonstrate and test the features of scheduling algorithms. In this domain, you can build test systems, view the generated code (comments) for multiple processors, and display a Gantt chart for parallel schedules. In derived domains, real code is generated, compiled, downloaded and executed, all under control of the selected target. In Ptolemy 0.7, one serious weakness of the code generation domains is that they only support scalar data types (complex, floating-point, integer, and fixed-point) on the input and output ports.

4.12 Code generation in C (CGC)

The CGC domain uses Boolean-controlled dataflow semantics, and has `C` as its target language. We have made every effort to name stars and their parameters consistently so that it is easy to move from one domain to another. With a little effort, one could create CGC versions of all SDF stars. If this were accomplished, then retargeting from one domain to another would be a simple matter of changing domains and targets and running the system again.

The generated `C` code is statically scheduled, and the memory used to buffer data between stars is statically allocated. Moreover, for many of the stars, the code that is generated depends on the values of the parameters. One way to think of this is that the parameters of the star are evaluated at code generation time, so no run-time overhead is incurred from the added flexibility of parameterizing the star.

There are several targets to choose from in the CGC domain. The `bdf-CGC` target supports the boolean-controlled dataflow model of computation. It must be used whenever stars with BDF semantics are present in a program graph. The `default-CGC` target supports the SDF model of computation, so it can be used when the program graph contains only stars with SDF semantics. The `TclTk_Target` target also supports SDF, and must be used whenever Tcl/Tk stars are present in the program graph. The `unixMulti_C` target supports SDF and partitions the program graph for execution on multiple workstations on a network.

4.13 Code generation for the Motorola DSP56000 (CG56)

This domain synthesizes assembly code for the Motorola DSP56000 family. The code generation techniques that are used are described in [Pin93]. They are derived from techniques used in Gabriel [Bie90]. We have used this domain to generate real-time implementations of various modem standards, touchtone generators, and touchtone decoders [Eva96], on an Ariel S-56X 560001 board.

4.14 Code generation in VHDL (VHDL, VHDLB)

This pair of domains is for generating code in VHDL (VHSIC Hardware Description Language). The VHDL domain supports functional models using the SDF model of computation, while VHDLB supports behavioral models using the native VHDL discrete event model of computation. Since the VHDL domain is based on the SDF model, it is independent of any notion of time. The VHDLB domain supports time delays and time-dependent behavior of blocks. The VHDL domain is intended for modeling systems at the functional block level, as in DSP functions for filtering and transforms, or in digital logic functions, independent of implementation issues. The VHDLB domain is intended for modeling the behavior of components and their interactions in system designs at all levels of abstraction.

Within the VHDL domain there are a number of different `Targets` to choose from. The default target, `default-VHDL`, generates sequential VHDL code in a single process within a single entity, following the execution order from the SDF scheduler. This code is suitable for efficient simulation, since it does not generate events on signals. The `SimVSS-VHDL` target is derived from `default-VHDL` and it provides facilities for simulation using the Synopsys VSS VHDL simulator. Communication actors and facilities in the `SimVSS-VHDL` target support code synthesis and co-simulation of heterogeneous CG systems under the `CompileCGSubsystems` target developed by José Luis Pino. There is also a `SimMT-VHDL` target for use with the Model Technology VHDL simulator. The `struct-VHDL` target generates VHDL code where individual actor firings are encapsulated in separate entities connected by VHDL signals. This target generates code which is intended for circuit synthesis. The `Synth-VHDL` target, derived from `struct-VHDL`, provides facilities for synthesizing circuit representations from the structural code using the Synopsys Design Analyzer toolset. Because the VHDL domain uses SDF semantics, it supports retargeting from other domains with SDF semantics (SDF, CGC, etc.) provided that the stars in the original graph are available in the VHDL domain. As this experimental domain evolves, more options for VHDL code generation from dataflow graphs will be provided. These options will include varying degrees of user control and automation depending on the target and the optimization goals of the code generation, particularly in VHDL circuit synthesis.

Unlike the VHDL domain, the older and less-developed VHDLB domain is much simpler in its operation. When a universe in the VHDLB domain is run, the graph is traversed and a codefile is generated in a pop-up window and in a subdirectory which reflects the topology and hierarchy of the graph. The generated VHDL code will reference VHDL entities which are expected to be included in other files. There is a VHDL codefile in the `$PTOLEMY/src/domains/vhdlb/lib` directory for each VHDL star in the main star palettes of the `$PTOLEMY/src/domains/vhdlb/icons` directory. Adding a new star is a matter of writing VHDL code for the entity and adding a star file in the stars subdirectory of the VHDLB domain which reflects the inputs, outputs, and parameters of that star. The existing stars should serve as examples for how new stars can be written.

Table 4-1 below summarizes the various domains

Domain	**Description**
Synchronous Data Flow (SDF)	• Oldest and most mature domain; it is a sub-domain of DDF, BDF, and PN. • Special case of data flow model of computation developed by Dennis. • Flow is completely predictable at compile time thus allows for efficient scheduling. • Allows for static scheduling. • Good match for synchronous signal processing systems with sample rates that are rational multiples of one another. • Supports multi-rate applications and has a rich star library. • Range of applications is limited.
Dynamic Data Flow (DDF)	• Versatile model of computation as it supports conditionals, data-dependent iteration, and true recursion. • More general than SDF. • Uses dynamic (run-time) scheduling which is more expensive than static scheduling. • Good match for signal processing applications with a limited amount of run-time control.
Boolean Data Flow (BDF)	• Relatively new domain which supports run-time flow of control. • Attempts to construct a compile-time schedule to try and achieve efficiency of SDF with generality of DDF. • More limited than DDF. • Constructs an annotated schedule: execution of a task is annotated with a boolean condition.
Integer and State Controlled Data Flow (STDF)	• Very new to Ptolemy and still experimental. • Realizes data flow control by integer control data and port statuses. It is an extension to BDF. • Scheduling is static and conditional like BDF. • It has user-defined evaluation functions.
Discrete Event (DE)	• Relatively mature domain which uses an event-driven model of computation. • Particles carry time-stamps which represent events that occur at arbitrary points in simulated time. • Events are processed in chronological order.
Finite State Machine (FSM)	• Very new to Ptolemy and still experimental. • Good match for control-oriented systems like real-time process controllers. • Uses a directed node-and-arc graph called a state transition diagram (STD) to describe the FSM.

Domain	Description
Higher Order Functions (HOF)	• Implements behavior of functions that may take a function as an argument and return a function. • HOF collection of stars may be used in all other domains. • Intended to be included only as a sub-domain by other domains.
Process Network (PN)	• Relatively new domain that implements Kahn process networks which is a generalization of data flow – processes replace actors. • Implements concurrent processes but without a model of time. • Uses POSIX threads. • SDF, BDF, and DDF are sub-domains of PN.
Multidimensional Synchronous Data Flow (MDSDF)	• Relatively new and experimental. • Extends SDF to multidimensional streams. • Provides ability to express a greater variety of dataflow schedules in a graphically compact way. • Currently only implements a two-dimensional stream.
Synchronous/Reactive (SR)	• Very new to Ptolemy and still experimental. • Implements model of computation based on model of time used in Esterel. • Good match for specifying discrete reactive controllers.
Code Generation (CG)	• Base domain from which all code generation domains are derived. • Supports a dataflow model that is equivalent to BDF and SDF semantics. • This domain only generates comments, allows viewing of the generated comments, and displays a Gantt Chart for parallel schedules. • Can only support scalar data types on the input and output ports. • All derived domains obey SDF semantics. • Useful for testing and debugging schedulers. • Targets include bdf-CGC which supports BDF, default-CGC which supports SDF semantics, TclTk_Target which supports SDF and must be used when Tcl/Tk stars are present, and unixMulti_C which supports SDF semantics and partitions the graph for multiple workstations on a network.
Code Generation in C (CGC)	• Uses data flow semantics and generates C code. • Generated C code is statically scheduled and memory used to buffer data between stars is statically allocated.
Code Generation for the Motorola DSP 56000 (CG56)	• Synthesizes assembly code for the Motorola DSP56000 family.

Domain	**Description**
Code Generation in VHDL (VHDL, VHDLB)	• Relatively new and experimental • Generates VHDL code. • VHDL domain supports SDF semantics whereas VHDLB supports behavioral models using native VHDL discrete event model of computation. • Many targets to choose from. • VHDL domain is good for modeling systems at functional block level whereas VHDLB is good for modeling behavior of components and their interactions at all levels of abstraction.

TABLE 4-1: Summary of the various Ptolemy domains.

The table below summarizes the various schedulers

Scheduler Name	**Features**
Default SDF Scheduler	• Performed at compile time. • Many possible schedules but schedule is chosen based on a heuristic that minimizes resource costs and amount of buffering required. • No looping employed so if there are large sample rate changes, size of generated code is large.
Joe’s Scheduler	• Performed at compile time. • Sample rates are merged wherever deadlock does not occur. • Loops introduced to match the sample rates. • Results in hierarchical clustering. • Heuristic solution so some looping possibilities are undetected.
SJS (Shuvra-Joe-Soonhoi) Scheduler	• Performed at compile time. • Uses Joe’s Scheduler at front end and then uses an algorithm on the remaining graph to find the maximum amount of looping available.
Acyclic Loop Scheduler	• Performed at compile time. • Constructs a single appearance schedule that minimizes amount of buffering required. • Only intended for acyclic dataflow graphs.

TABLE 4-2: Summary of Uniprocessor schedulers

Table 4-3 below summarizes the multiprocessor schedulers.

Scheduler Name	Features
Hu’s Level-based List Scheduler	• Performed at compile time. • Most widely used. • Tasks assigned priorities and placed in a list in order of decreasing priority. • Ignores communication costs when assigning functional blocks to processors.
Sih’s Dynamic Level Scheduler	• Performed at compile-time. • Assumes that communication and computation can be overlapped. • Accounts for interprocessor communication overheads and interconnection topology.
Sih’s Declustering Scheduler	• Performed at compile-time. • Addresses trade-off between exploiting parallelism and interprocessor communication overheads. • Analyzes a schedule and finds the most promising placements of APEG nodes. • Not single pass but takes an iterative approach.
Pino’s Hierarchical Scheduler	• Performed at compile time. • Partially expands the APEG. • Can use any of the above parallel schedulers as a top-level scheduler. • Supports user-specified clustering. • Realizes multiple orders of magnitude speedup in scheduling time and reduction in memory usage.

TABLE 4-3: Summary of multiprocessor schedulers.

4.15 Domains that have been removed

This section highlights the experimental domains that were removed from the Ptolemy distribution. If any users are interested in resurrecting these domains, then please send e-mail to `ptolemy@ptolemy.eecs.berkeley.edu`.

4.15.1 Circuit simulation (Thor)

Like DE, the Thor domain was event-driven. However, it was specialized to register-transfer level simulation of digital circuits. It was based on the Thor simulator developed at Stanford, which in turn was based on a simulation kernel developed at the University of Colorado. The domain was written by Suengjun Lee. Its capabilities were similar to a variety of commercial circuit simulators. The Thor domain was based on very old circuit simulation technology. Contemporary equivalents include VHDL and Verilog simulators. The VHDL domains thus replace Thor, at least in part, although currently the star library is not as rich.

4.15.2 Communicating processes (CP)

The CP domain, developed by Suengjun Lee and based on thread classes developed by Thomas M. Parks, modeled multiple simultaneous processes with independent flow of control. It was based on the Sun lightweight process library. Because of this dependence on proprietary code, it was only available on Sun workstations. CP was a timed domain. Processes communicated by exchanging particles with time stamps. The particles are processed in chronological order, in that the process with the oldest time stamps at its inputs is activated. From the perspective of the star writer, the star is always running, presumably in an infinite loop, responding to input events and producing output events. Because of this model, the domain was well suited to high-level modeling of distributed systems. Each component in the system would be represented as a program that appears to run as an autonomous agent, concurrently with other components of the system.

The CP domain is probably the most useful of the domains that have been removed. The problem is that it is based on the Sun lightweight process library, which Sun Microsystems is no longer supporting. The Lightweight Processes library does not run on recent releases of the Solaris operating systems. We took a stab at porting the domain to use Posix threads, the modern replacement, but this task overwhelmed the resources we had available. We would be very interested in volunteers interested in pursuing this.

4.15.3 Message queueing (MQ)

The MQ domain was based on an object-oriented approach for software development developed by E. C. Arnold and D. W. Brown at AT&T Bell Laboratories. The run-time environment viewed components as addressable objects capable of receiving messages, sending messages, and maintaining an individual state managed solely by the component’s methods. Each message contained sufficient information for its destination object to perform appropriate updating of internal data structures and to produce other messages.

By applying this model in Ptolemy, stars would pass message particles to one another. Connections between pairs of stars are bidirectional so that client-server relationships can be established over these links. From the name of this domain, it can be understood that messages sent from a particular star to another were always processed in sequence. However, the execution order of stars in the system is arbitrary, and a star, when fired, steps through its portholes in an arbitrary fashion as well, processing the first incoming message arriving at each port, should one exist.

The MQ domain was well-suited to the development of call-processing software. Its use in the modeling of system control was illustrated in a sophisticated cell-relay network simulation. This large-scale, heterogeneous demo used the SDF domain to specify space-division packet switching fabrics, the DE domain to model “timed” network subsystems, and the MQ domain to describe a centralized network controller.

Although it introduced a number of interesting features, this domain did not find wide usage, so it has been removed.

4.15.4 Code generation for the Sproc multiprocessor DSP (Sproc)

The Sproc multiprocessor DSP used to be made by Star Semiconductor [Mur93], however, neither the processor nor the company now exist, so this domain has been removed.

4.15.5 Code generation for the Motorola DSP96000 (CG96)

This domain is similar to CG56, except that the synthesized language is assembly code for the Motorola DSP96000 (floating-point) family. This processor is no longer being developed or improved by Motorola, so we have removed this domain.

4.15.6 Code generation in Silage (Silage)

This was a code generation domain for the Silage language. Silage is an applicative functional textual language that was developed to describe fixed-point digital signal processing (DSP) algorithms, which are well-suited for hardware implementation. Silage descriptions serve as the input specification for some high-level synthesis tools (e.g. Hyper from U.C. Berkeley, Cathedral from IMEC, and the DSP Station from Mentor Graphics). Asawaree Kalavade used the Ptolemy interface to Hyper to estimate costs of hardware implementations in hardware/software codesign experiments [Kal93,Kal94,Kal96]. Berkeley, however, is moving away from Silage, and there appears to be little future for it, so we have removed this domain.

4.15.7 Functional Code Generation in VHDL (VHDLF)

The VHDLF domain was originally intended to contrast with the VHDLB domain. It supported structural code generation using VHDL blocks with no execution delay or timing behavior, just functionality. The semantics for the VHDLF domain were not strictly defined, and the scheduling depended on how the underlying VHDL code blocks associated with each VHDLF star were written. The VHDLF domain has been replaced by the VHDL domain. The VHDL domain is not meant to be used in the same way as the VHDLF domain, however. The VHDL domain is for generating code from functional block diagrams with SDF semantics.

4.16 Interfaces to Foreign Tools

The Ptolemy design environment is a collection of dozens of collaborating tools with interfaces to dozens of others, as shown in Figure 4-4. Within Ptolemy there are many different domains, schedulers, and targets, each of which is a tool in its own right. Those tools are derived from a common framework provided by the Ptolemy kernel. Other tools, such as an expression evaluator for parameter expressions and filter design programs, are also embedded.

Not every Ptolemy interface is listed in Figure 4-4. We have written several targets and domains that we have not released to the public. For example, we are developing a CGC target for the UltraSparc Visual Instruction Set. We have developed but never released code generation domains for the AT&T DSP 3 multiprocessor system and the Philips Video Signal Processor system [Shi94]. We have eliminated several domains, as listed in Section 4.15, such as a code generation for the Sproc multiprocessor DSP system [Mur93].

This rest of this section focuses on Ptolemy interfaces to foreign tools that are not included in the Ptolemy release. These foreign tools are standalone programs, such as compilers, assemblers, interpreters, and simulators.

4.16.1 Specification and Layout

Defining systems, subsystems, blocks, and connections can be expressed graphically using pigi (see Chapter 2) and textually using ptcl (see Chapter 3). Graphical descriptions can be converted to textual specifications. The two interfaces can work together. By running pigi with the -console option, one can evaluate ptcl commands in the pigi console. The pigi run-

Released with Ptolemy	*System Design*	*Not Released with Ptolemy*
oct/vem higher-order functions ptcl Tycho	Specification and Layout	Emacs, vi, xedit
expression evaluator Tcl	Parameter Calculation	MATLAB Mathematica
simulation domains filter designer	Algorithm Prototyping	MATLAB Utah Raster Toolkit Esterel
Tcl/Tk, Itcl/Tk pxgraph, xv	Display and Visualization	MATLAB soundtool, audiotool
simulation domains multirate schedulers run-time schedulers wormholes	Simulation	sim56000, sim96000 Synopsys VSS Model Technology VSIM IPUS, ArrayOL domains
code generation domains gcc, g++, gmake parallel schedulers hierarchical scheduler	Synthesis	asm56000, Ariel S-56X card commercial C/C++ compilers Synopsys Design Analyzer

TABLE 4-4: Ptolemy interfaces to various tools.

control panel allows control of runs by ptcl scripts. Blocks exists that execute ptcl scripts.

Schematic entry is implemented by vem, and schematics are databased in oct. Scalable systems can be specified graphically using higher-order functions. Tycho provides a language-sensitive editor and an evolving framework for future graphical user interfaces.

4.16.2 Parameter Calculation

Parameter calculation maps system parameters into parameters in the subsystems, blocks, and connections. The calculation is controlled by an expression evaluator, which supports calls to ptcl. Because ptcl has interfaces to MATLAB and Mathematica, MATLAB and Mathematica expressions can be embedded in parameter specifications.

4.16.3 Algorithm Prototyping and Visualization

A designer can develop domain-specific algorithms in Ptolemy, such as for speech coding. For filter design, one can use MATLAB or Ptolemy's filter design programs. A variety of Unix and X windows utilities are used for display and visualization of data.

4.16.4 Simulation

Ptolemy provides the ability to simulate complex systems. The key is the notion of a wormhole that allows a designer to mix domain-specific algorithms together to cosimulate the functionality or behavior of the system. wormholes & hierarchical scheduling

4.16.5 Synthesis

Ptolemy provides mature abilities to synthesis dataflow systems. From dataflow graphs, Ptolemy can generate C, C++, Motorola 56000 assembly code, and VHDL for uniprocessor and multi-processor systems.

Ptolemy provides the ability to synthesis complex systems. The key is the notion of hierarchical scheduling that allows multiple implementation technologies to cosimulate. Mixing this with wormholes allows simulation domains to participate in cosimulation. This combinations allows a complex system to be simulated at a variety of levels of detail.

Chapter 5. SDF Domain

Authors: *Shuvra Bhattacharyya*
Joseph T. Buck
Michael J. Chen
Brian L. Evans
Soonhoi Ha
Paul Haskell
Christopher Hylands
Alan Kamas
Alireza Khazeni
Bilung Lee
Edward A. Lee
David G. Messerschmitt

Other Contributors: *Asawaree Kalavade*
Thomas M. Parks
Gregory S. Walter

5.1 Introduction

Synchronous dataflow (SDF) is a data-driven, statically scheduled domain in Ptolemy. It is a direct implementation of the techniques given in [Lee87a] and [Lee87b]. “Data-driven” means that the availability of `Particles` at the inputs of a star enables it. Stars without any inputs are always enabled. “Statically scheduled” means that the firing order of the stars is determined once, during the start-up phase. The firing order will be periodic. The SDF domain is one of the most mature in Ptolemy, having a large library of stars and demo programs. It is a simulation domain, but the model of computation is the same as that used in most of the code generation domains. A number of different schedulers, including parallel schedulers, have been developed for this model of computation.

5.1.1 Basic dataflow terminology

SDF is a special case of the dataflow model introduced by Dennis [Den75]. It is equivalent to the *computation graph* model of Karp and Miller [Kar66]. In the terminology of the dataflow literature, stars are called *actors*. An invocation of the `go()` method of a star is called a *firing*. Particles are called *tokens*. In a digital signal processing system, a sequence of tokens might represent a sequence of samples of a speech signal or a sequence of frames in a video sequence.

When an actor fires, it consumes some number of tokens from its input arcs, and produces some number of output tokens. In synchronous dataflow, these numbers remain constant throughout the execution of the system. It is for this reason that this model of computation is suitable for synchronous signal processing systems, but not for asynchronous systems. The fact that the firing pattern is determined statically is both a strength and a weakness of this domain. It means that long runs can be very efficient, a fact that is heavily exploited in the code generation domains. But it also means that data-dependent flow of control is not allowed. This would require dynamically changing firing patterns. The Dynamic Dataflow (DDF) and Boolean Dataflow (BDF) domains were developed to support this, as described in chapters 7 and 8, respectively.

5.1.2 Balancing production and consumption of tokens

Each porthole of each SDF star has an attribute that specifies the number of particles consumed (for input ports) or the number of particles produced (for output ports). When you connect two portholes with an arc, the number of particles produced on the arc by the source star may not be the same as the number of particles consumed from that arc by the destination star. To maintain a balanced system, the scheduler must fire the source and destination stars with different frequency.

Consider a simple connection between three stars, as shown in figure 5-1. The symbols adjacent to the portholes, such as N_{A1}, represent the number of particles consumed or produced by that porthole when the star fires. For many signal processing stars, these numbers are simply one, indicating that only a single token is consumed or produced when the star fires. But there are three basic circumstances in which these numbers differ from one:

- Vector processing in the SDF domain can be accomplished by consuming and producing multiple tokens on a single firing. For example, a star that computes a fast Fourier transform (FFT) will typically consume and produce 2^M samples when it fires, where M is some integer. Examples of vector processing stars that work this way are `FFTCx`, `Average`, `Burg`, and `LevDur`. This behavior is quite different from the matrix stars, which operate on particles where each individual particle represents a matrix.
- In multirate signal processing systems, a star may consume M samples and produce N, thus achieving a sampling rate conversion of N/M. For example, the `FIR` and `FIRCx` stars optionally perform such a sampling rate conversion, and with an appropriate choice of filter coefficients, can interpolate between samples. Other stars that perform sample rate conversion include `UpSample`, `DownSample`, and `Chop`.
- Multiple signals can be merged using stars such as `Commutator` or a single signal can be split into subsignals at a lower sample rate using the `Distributor` star.

To be able to handle these circumstances, the scheduler first associates a simple balance equation with each connection in the graph. For the graph in figure 5-1, the balance equations are

$$r_A N_{A1} = r_C N_{C1}$$
$$r_A N_{A2} = r_B N_{B1}$$

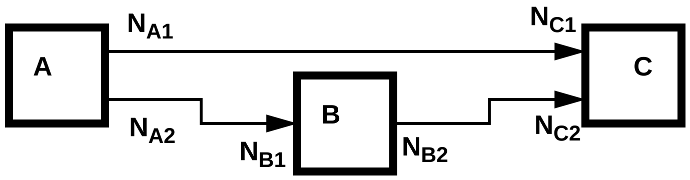


FIGURE 5-1: A simple connection of SDF stars, used to illustrate the use of balance equations in constructing a schedule.

$$r_B N_{B2} = r_C N_{C2}$$

This is a set of three simultaneous equations in three unknowns. The unknowns, r_A, r_B, and r_C are the *repetitions* of each actor that are required to maintain balance on each arc. The first task of the scheduler is to find the smallest non-zero integer solution for these repetitions. It is proven in [Lee87a] that such a solution exists and is unique for every SDF graph that is “consistent,” as defined below.

5.1.3 Iterations in SDF

When running an SDF system under the graphical user interface, you will have the opportunity to specify “when to stop.” Since the SDF domain has no notion of time, this is not given in units of time. Instead, it is given in units of SDF iterations. At each SDF iteration, each star is fired the minimum number of times to satisfy the balance equations.

Suppose for example that star B in figure 5-1 is an `FFTCx` star with its parameters set so that it will consume 128 samples and produce 128 samples. Suppose further that star A produces exactly one sample on each output, and star C consumes one sample from each input. In summary,

$$N_{A1} = N_{A2} = N_{C1} = N_{C2} = 1$$
$$N_{B1} = N_{B2} = 128.$$

The balance equations become

$$r_A = r_C$$
$$r_A = 128 r_B$$
$$128 r_B = r_C.$$

The smallest integer solution is

$$r_A = r_C = 128$$
$$r_B = 1.$$

Hence, each iteration of the system includes one firing of the `FFTCx` star and 128 firings each of stars A and B.

5.1.4 Inconsistency

It is not always possible to solve the balance equations. Suppose that in figure 5-1 we have

$$N_{A1} = N_{A2} = N_{C1} = N_{C2} = N_{B1} = 1$$
$$N_{B2} = 2.$$

In this case, the balance equations have no non-zero solution. The problem with this system is that there is no sequence of firings that can be repeated indefinitely with bounded memory. If we fire A,B,C in sequence, a single token will be left over on the arc between B and C. If we repeat this sequence, two tokens will be left over. Such a system is said to be *inconsistent*, and is flagged as an error. The SDF scheduler will refuse to run it. If you must run such a system, change the domain of your graph to the DDF domain.

5.1.5 Delays

Delays are indicated in Pigi by small green diamonds that are placed on an arc. Most of the standard palettes of stars have the delay icon at the upper left. The delay has a single parameter, the number of samples of delay to be introduced. In the SDF domain, a delay with parameter equal to one is simply an initial particle on an arc. This initial particle may enable a star, assuming that the destination star for the delay arc requires one particle in order to fire. To avoid deadlock, all feedback loops much have delays. The SDF scheduler will flag an error if it finds a loop with no delays. For most particle types, the initial value of a delay will be zero. For particles which hold matrices, the initial value is an empty Envelope, which must be checked for by stars which work on matrix inputs. Initializable delays allow the user to give values to the initial particles placed in the buffer. Please refer to 2.12.8 on page 2-47 for details on how to use initializable delays.

5.2 An overview of SDF stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all of the installed domains. For the SDF domain, the star library is large enough that it has been divided into sub-palettes. The top-level palette is shown in figure 5-2

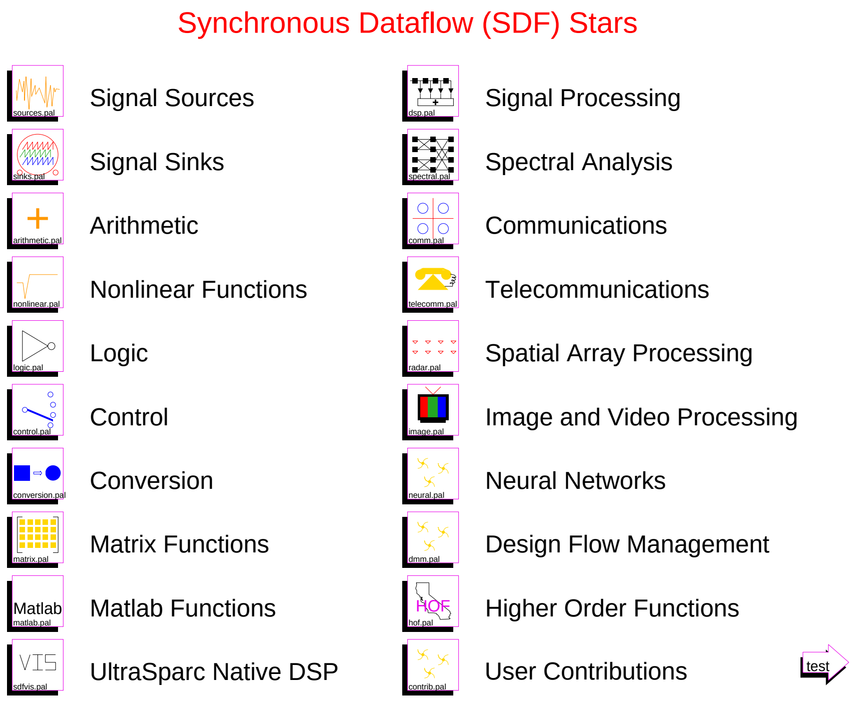


FIGURE 5-2: The top-level palette for accessing the library of SDF stars.

The "sources" palette contains signal generators of various types. The "sinks" palette contains various stars that display signals in different ways or write the value of signal samples to files. The "arithmetic" palette contains basic adders, subtracters, multipliers, and amplifiers, for all the standard scalar data types (floating point, complex, fixed-point, and integer). The “nonlinear” palette contains stars that compute transcendental functions, such as logarithm, cosine, sine, and exponential functions, as well as quantizer and table lookup stars. The “logic” palette contains stars that perform Boolean and comparison operations, such as and, or, and greater than. The “control” palette contains stars that manipulate the flow of tokens, such as commutators and distributors, downsamplers and upsamplers, and forks. The “conversion” palette contains stars that explicitly accomplish type conversion. The “matrix” palette contains matrix operators such as matrix addition and multiplication. More complex stars that use matrix operations internally can be found in other palettes, such as the singular value decomposition and Kalman filters in the “dsp” palette. The “matlab” palette contains stars that communicate with a Matlab process and thus have access to all of the functionality of Matlab. The “vis” palette contains stars that use the Sun UltraSparc Visual Instruction Set.The “dsp” palette contains various signal processing functions such as fixed and adaptive filters of various types. The “spectral” palette contains spectral estimation functions. The “communications” palette contains stars that are specific to digital communications functions, such as pulse shapers, speech coders, and QAM encoders. The “telecommunications” palette contains touchtone generators and decoders, channel models, and PCM coders. The “spatial array palette” contains models of sensors, Doppler effects, and beamformers. The “image” palette contains stars for image and video signal processing. The “neural” palette contains neural network stars. The “dfm” palette contains design flow management stars that use strings and files as datatypes. The “hof” palette contains the Higher Order Functions available in the SDF domain. The HOF stars in this palette are explained in detail in the HOF domain chapter. The “user” palette contains user contributed stars.

Each palette is summarized in more detail below. In the listing, whenever data types are not mentioned, double-precision floating point is used. Not all data types are represented in all stars. Type conversions, automatic or explicit, can be used to complete the collection.

The parameters of a star are shown in italics. More information about each star can be obtained using the on-line `profile` command (“`,`”), or the on-line `man` command (“`M`”).

At the top of each palette, for convenience, are instances of the two delay icons, the bus icon, and the following star:

`BlackHole`	Discard all inputs. This star is useful for discarding signals that are not useful.

The delay and bus icons are created on top of an arc to define its properties and are not stars.

5.2.1 Source stars

Source stars are stars with no inputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. In the dataflow model of computation, they are always enabled, and hence can be fired at any time. In the synchronous dataflow model, the frequency with which they are fired, relative to other stars in the system, is determined by the solution to the balance equations. The palette of source stars is shown in figure 5-3, and the stars are summarized below, in the order they appear in the palette.

Floating-point sources

`Const`	Output a constant signal with value given by the *level* parameter

	(default 0.0).
`DTMFGenerator`	Create a dual-tone modulated-frequency signal, such as the tone generated by a touchtone telephone.
`Impulse`	Generate a single impulse or an impulse train. Each impulse has an amplitude *level* (default 1.0). If *period* (default 0) is equal to 0, then only a single impulse is generated; otherwise, *period* specifies the period of the impulse train.
`IIDGaussian`	Generate an identically independently distributed white Gaussian pseudo-random process with *mean* (default 0) and *variance* (default 1).
`IIDUniform`	Generate an identically independently distributed uniformly distributed pseudo-random process. Output is uniformly distributed between *lower* (default 0) and *upper* (default 1).
`Ramp`	Generate a ramp signal, starting at *value* (default 0.0) and incrementing by step size *step* (default 1.0) on each firing.
`RanConst`	Generate an random number with a uniform(u), exponential(e), or normal(n) distribution, as determined by the *distribution*

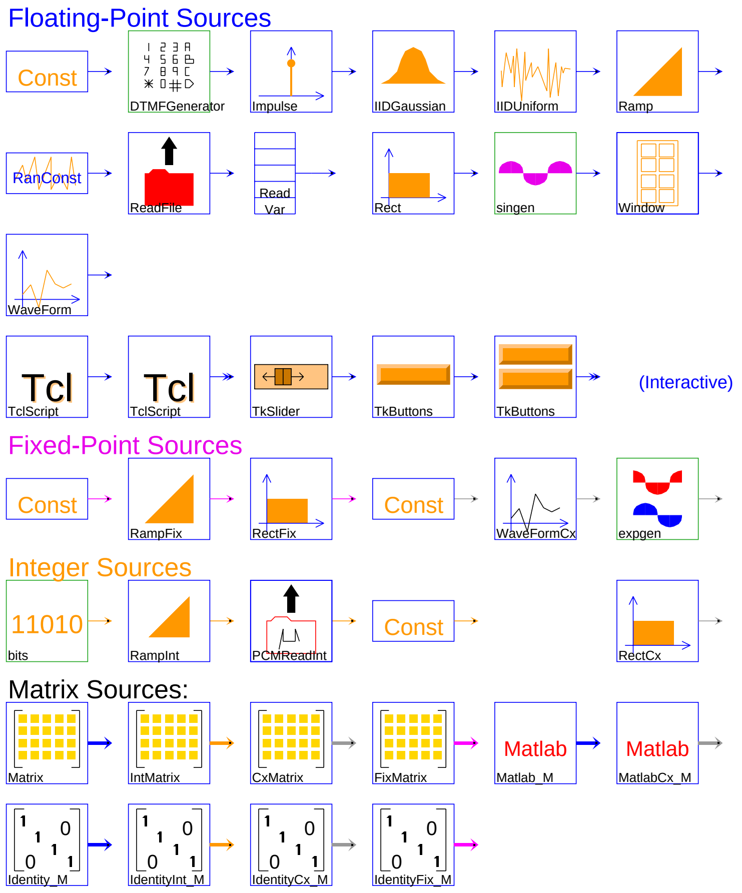


FIGURE 5-3: The palette of source stars for the SDF domain.

parameter. This star is new in Ptolemy 0.7.

`ReadFile` Read ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.

`ReadVar` Output the value of a double-precision floating point variable from a shared memory. Use the `writeVar` star to write values into the shared memory.
WARNING: This star may produce unpredictable results, since the results will depend on the precendences in the block diagram in which it appears as well as the scheduler used.

`Rect` Generate a rectangular pulse of *height* (default 1.0) and *width* (default 8). If *period* is greater than zero, then the pulse is repeated with the given period.

`singen` Generate a sine wave with *frequency* (relative to the given *sample_rate*) and phase given by *phase_in_radians*. This is implemented as a galaxy according to the formula

$$\sin(2 \pi n \textit{ frequency} / \textit{sample\_rate} + \textit{phase\_in\_radians})$$

where n is the sample index. Therefore, *frequency* and *sample_rate* must have the same units, e.g. rad/sample, Hz, etc.

`WaveForm` Output a waveform as specified by the array state *value* (default “1 -1”). You can get periodic signals with any period, and can halt a simulation at the end of the given waveform. The following table summarizes the capabilities:

haltAtEnd	*periodic*	*period*	*operation*
NO	**YES**	**0**	**The period is the length of the waveform**
NO	**YES**	**N>0**	**The period is N**
NO	**NO**	**anything**	**Output the waveform once, then zeros**
YES	**anything**	**anything**	**Stop after outputting the waveform once**

The first line of the table gives the default settings. This star may be used to read a file by simply setting *value* to something of the form `< filename`, preferably specifying a complete path.

`Window` Generate standard window functions or periodic repetitions of standard window functions. The possible functions are: `Rectangle`, `Bartlett`, `Hanning`, `Hamming`, `Blackman`, `Kaiser` and `SteepBlackman`. One period of samples is produced at each firing.

`TclScript` (Two icons) Invoke a Tcl script that can optionally define a procedure that is invoked every time the star fires. That procedure can read the star’s inputs and update the value of the outputs.

`TkSlider`	Output a value determined by an interactive on-screen scale slider.
`TkButtons`	This star outputs the value 0.0 on all outputs unless the corresponding button is pushed. When the button is pushed, the output takes the value given by the parameter *value*. If *synchronous* is `YES`, then outputs are produced only when some button is pushed. I.e., the star waits for a button to be pushed before its `go` method returns. If *allow_simultaneous_events* is `YES`, then the buttons pushed are registered only when the button labeled "PUSH TO PRODUCE OUTPUTS" is pushed. Note that if *synchronous* is `NO`, this star is nondeterminate.

Fixed-point sources

`ConstFix`	Constant source for fixed-point values.
`RampFix`	Ramp for fixed-point values.
`RectFix`	Generate a fixed-point rectangular pulse of *height* (default 1.0). and *width* (default 8). If *period* is greater than zero, then the pulse is repeated with the given period. The precision of *height* can be specified in bits.

Complex sources

`ConstCx`	Constant source for complex values.
`WaveFormCx`	Output a complex waveform as specified by the array state *value* (default "(1,0) (-1,0)"). Note that "(a,b)" means a + b j. The parameters work the same way as in the `WaveForm` star.
`expgen`	Generate a complex exponential with the given frequency (relative to the *sample_rate* parameter).
`RectCx`	Generate a rectangular pulse of *height* (default 1.0) and *width* (default 8). If *period* is greater than zero, then the pulse is repeated with the given period.Integer sources
`bits`	Produce "0" with probability *probOfZero*, else produce "1".
`RampInt`	Ramp for integer values.
`PCMReadInt`	Read a binary μ-law encoded PCM file. Return one sample on each firing. The file format that is read is the same as the one written by the `Play` star. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros. This star is new in Ptolemy 0.7.
`ConstInt`	Constant source for integer values.

Matrix Sources

The `Matrix` and `Identity` stars each have four different icons for the different matrix data types.

`Matrix`	(four icons) Produce a matrix with floating-point entries. The entries are read from the array parameter *FloatMatrixContents* in rasterized order: i.e., for a $M \times N$ matrix, the first row is filled from left to right using the first N values from the array.
`Matlab_M`	Evaluate a Matlab function if inputs are given or evaluate a Matlab command if no inputs are given. Any Matlab script can be evaluated, provided that the current machine has a license to run Matlab. See “Matlab stars” on page 5-26.
`MatlabCx_M`	Complex version of the above star.
`Identity_M`	(four icons) Output a floating-point identity matrix.

5.2.2 Sink stars

The stars in the palette of figure 5-4 are those with no outputs. They display signals in various ways, or write them to files.

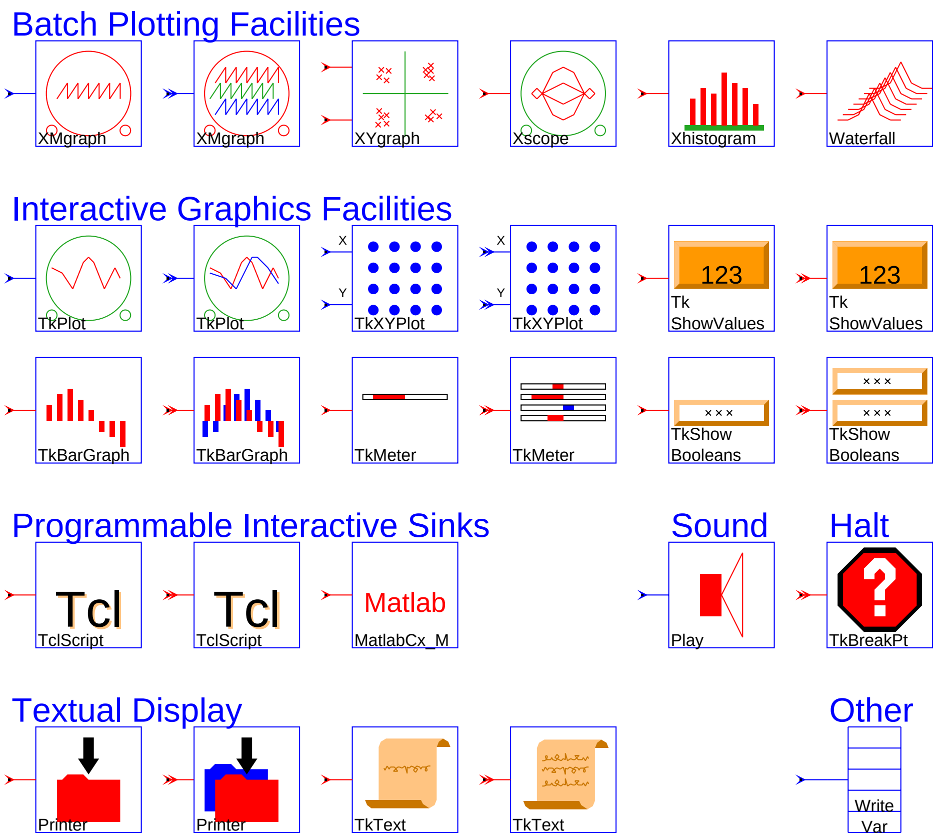


FIGURE 5-4: Sink stars in the SDF domain.

Batch Plotting Facilities

The first six stars in this palette are all based on the `pxgraph` program. This program has many options, summarized in “pxgraph — The Plotting Program” on page 20-1. The differences between stars often amount to little more than the choice of default options. Some, however, preprocess the signal in useful ways before passing it to the `pxgraph` program. The first allows only one input signal, the second allows any number (notice the double arrow on the input port).

`XMgraph`	(two icons) Generate a generic multi-signal plot.
`XYgraph`	Generate an *X*-*Y* plot with the `pxgraph` program. The *X* data is on “xInput” and the *Y* data is on “input”.
`Xscope`	Generate a multi-trace plot with the `pxgraph` program. Successive traces are overlaid on one another.
`Xhistogram`	Generate a histogram with the `pxgraph` program. The parameter *binWidth* determines the bin width.
`Waterfall`	Plot a series of traces in the style of a waterfall plot. This is a type of three-dimensional plot used to show the evolution of signals or spectra. Optionally, each plot can be made opaque, so that lines that would appear behind the plot are eliminated.

Interactive Graphics Facilities

These stars are multiple configurations of only six stars. These stars all use the Tk toolkit associated with the Tcl language to create interactive, animated displays on the screen.

`TkPlot`	(two icons) Plot “Y” input(s) vs. time with dynamic updating. Two styles are currently supported: `dot` causes individual points to be plotted, whereas `connect` causes connected lines to be plotted. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.
`TkXYPlot`	(two icons) Plot “Y” input(s) vs. “X” input(s) with dynamic updating. Two styles are currently supported: `dot` causes points to be plotted, whereas `connect` causes connected lines to be plotted. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.
`TkShowValues`	(two icons) Display the values of the inputs in textual form. The print method of the input particles is used, so any data type can be handled, although the space allocated on the screen may need to be adjusted.
`TkBarGraph`	(two icons) Dynamically display the value of any number of input signals in bar-chart form. The first 12 input signals will be assigned distinct colors. After that, the colors are repeated. The colors can be controlled using X resources.

`TkMeter`	(two icons) Dynamically display the value of any number of input signals on a set of bar meters.
`TkShowBooleans`	(two icons) Display input Booleans using color to highlight their value.

Programmable Interactive Sinks

`TclScript`	(two icons) Invoke a Tcl script that can optionally define a procedure that is invoked every time the star fires. That procedure can read the star's inputs and update the value of the outputs.
`MatlabCx_M`	Evaluate a Matlab function if inputs are given or evaluate a Matlab command if no inputs are given.

Sound

`Play`	Play an input stream on the workstation speaker. This star works best on Suns, but can work on SGI Indigos and HP 700s and 800s. On HPs, you may need other publicly available software for this star to work. The *gain* parameter (default 1.0) multiplies the input stream before it is μ-law compressed and written. The inputs should be in the range of -32000.0 to 32000.0. The file is played at a fixed sampling rate of 8000 samples per second. When the wrapup method is called, a file of 8-bit μ-law samples is handed to a program named `ptplay` which plays the file. The `ptplay` program must be in your path. See"Sounds" on page 2-38 for more information.

Halt

`TkBreakPt`	A conditional break point. Each time this star executes, it evaluates its conditional expression. If the expression evaluates to true, it causes the run to pause.

Textual Display

`Printer`	(two icons) Print out one sample from each input port per line. The *fileName* parameter specifies the file to be written; the special names `<stdout>` and `<cout>` which specify the standard output stream, as well as `<stderr>` and `<cerr>` which specify the standard error stream, are also supported.
`TkText`	(two icons) Display the values of the inputs in a separate window, keeping a specified number of past values in view. The print method of the input particles is used, so any data type can be handled.

Other

`WriteVar`	Write the value of the input to a double-precision floating-point

	variable in shared memory. Use the `ReadVar` star to read values from the shared memory. WARNING: This star may produce unpredictable results, since the results will depend on the precedences in the block diagram in which it appears, as well as the scheduler (target) used.

5.2.3 Arithmetic stars

In principle, it should be possible to overload the basic arithmetic operators so that, for example, a single `Add` star could handle any data type. Our decision, however, was in favor of more explicit typing, in which there is an `Add` star for each particle type supported in the kernel. As before, when there is no data type suffix in the name of the star, the data type supported is double-precision floating point.

Many of the stars in this palette have more than one icon, as indicated in figure 5-5. Each such icon has a different configuration of ports. This is done for visual clarity in schematics. A port with a double arrowhead can accept any number of input signals. Each four rows of the palette contains equivalent stars for floating-point, complex, fixed-point, and integer arithmetic, respectively. Listed by the roots of the names of the stars, they are:

`Add`	(two icons) Output the sum of the inputs.
`Sub`	Output the "pos" input minus all "neg" inputs.
`Mpy`	(two icons) Output the product of the inputs.
`Gain`	This is an amplifier; the output is the input multiplied by the *gain* (default 1.0).

The floating-point and complex-valued scalar data types also have the following star:

`Average`	Average some number of input samples or blocks of input sam-

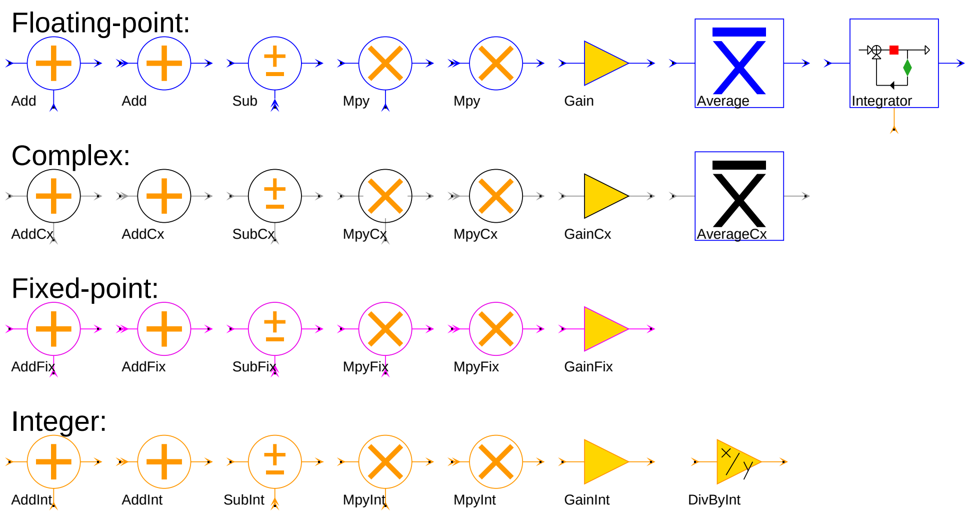


FIGURE 5-5: The arithmetic palette in the SDF domain. Note that several of the stars have more than one icon, each with a different configuration of ports.

ples. Blocks of successive input samples are treated as vectors.

The floating-point type has one additional arithmetic star:

`Integrator` This is an integrator with leakage, limits, and reset. With the default parameters, input samples are simply accumulated, and the running sum is the output. To prevent any resetting in the middle of a run, connect a `Const` source with value 0 to the “reset” input. Otherwise, whenever a non-zero is received on this input, the accumulated sum is reset to the current input (i.e. no feedback).

Limits are controlled by the *top* and *bottom* parameters. If *top* ≤ *bottom*, no limiting is performed (this is the default). Otherwise, the output is kept between *bottom* and *top*. If *saturate* = `YES`, saturation is performed. If *saturate* = `NO`, wrap-around is performed (this is the default). Limiting is performed before output.

Leakage is controlled by the *feedbackGain* parameter (default 1.0). The output is the data input plus *feedbackGain* × *state*, where *state* is the previous output.

The integer type has the following star:

`DivByInt` This is an amplifier. The integer “output” is the integer “input” divided by the integer *divisor* (default 1). Truncated integer division is used.

5.2.4 Nonlinear stars

The nonlinear palette (figure 5-6) in the SDF domain includes transcendental functions, quantizers, table lookup stars, and miscellaneous nonlinear functions.

Quantizers

`AdaptLinQuant` Quantize the input to one of 2^*bits* possible output levels. The high and low output levels are anti-symmetrically arranged around zero and their magnitudes are determined by (2^*bits*-1)*“inStep”/2. The steps between levels are uniformly spaced at the step size given by the “inStep” input value. The linear quantizer can be made adaptive by feeding back past information such as quantization level, quantization value, and step size into the current step size.

`LinQuantIdx` Quantize the input to the number of levels given by the *levels* parameter. The quantization levels are uniformly spaced between *low* and *high* inclusive. Rounding down is performed, so that output level will equal *high* only if the input level equals or exceeds *high*. If the input is below *low*, then the quantized output will equal *low*. The quantized value is output to the “amplitude” port, while the index of the quantization level is

	output to the “stepNumber” port.
`Quant`	Quantize the input value to one of *N*+1 possible output levels using *N* thresholds. For an input less than or equal to the n-th threshold, but larger than all previous thresholds, the output will be the n-th level. If the input is greater than all thresholds, the output is the *N*+1-th level. If level is specified, there must be one more level than thresholds; the default value for level is 0, 1, 2, ... *N*. This star is much slower than `LinQuantIdx`, so if possible, that one should be used instead.
`QuantIdx`	Quantize the input value to one of *N*+1 possible output levels using *N* thresholds, and output both the quantized result and the quantization level. See the `Quant` star for more information.
`Quantizer`	This star quantizes the input value to the nearest output value in the given codebook. The nearest value is found by a full search of the codebook, so the star will be significantly slower than either `Quant` or `LinQuantIdx`. The absolute value of the difference is used as a distance measure.

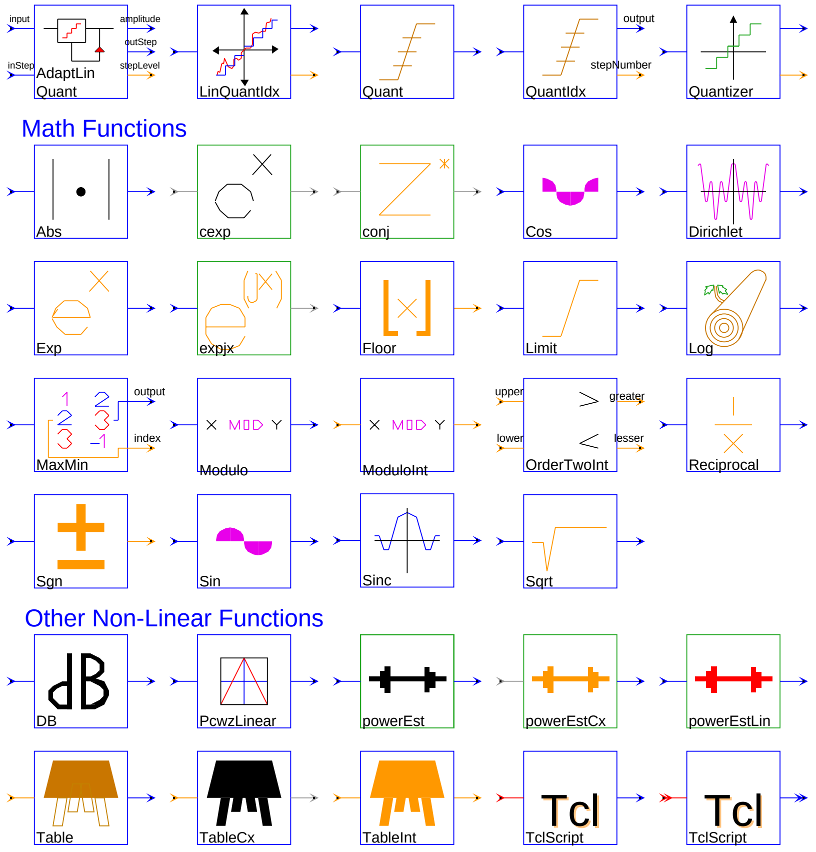


FIGURE 5-6: Palette of nonlinear stars for the SDF domain.

Math Functions

`Abs`	Compute the absolute value of its input.
`cexp`	Compute the complex exponential function of its complex input. See also `expjx`.
`conj`	Compute the conjugate of its complex input.
`Cos`	Compute the cosine of its input, assumed to be an angle in radians.
`Dirichlet`	Compute the normalized Dirichlet kernel (also called the aliased sinc function): $d_N(x) = \frac{\sin(Nx/2)}{N\sin(x/2)}$ The value of the normalized Dirichlet kernel at $x = 0$ is always 1, and the normalized Dirichlet kernel oscillates between -1 and $+1$. The normalized Dirichlet kernel is periodic in x with a period of either 2π when N is odd or 4π when N is even.
`Exp`	Compute the real exponential function of its real input.
`expjx`	Compute the complex exponential function of its real input. See also `cexp`.
`Floor`	Output the greatest integer less than or equal to its input.
`Log`	Output the natural logarithm of its input.
`Limit`	The output of this star is the value of the input limited to the range between *bottom* and *top* inclusive.
`MaxMin`	Finds maximum or minimum, value or magnitude, of a fixed number of data values on its input. If you want to use this star to operate over multiple data streams, then precede this star with a `Commutator` and set the parameter *N* accordingly.
`Modulo`	The output is equal to the remainder after dividing the input by the *modulo* parameter.
`ModuloInt`	The output is equal to the integer remainder after dividing the integer input by the integer *modulo* parameter.
`OrderTwoInt`	Takes two inputs and outputs the greater and lesser of the two integers.
`Reciprocal`	Output the reciprocal of its input, with an optional magnitude limit. If the magnitude limit is greater than zero, and the input value is zero, then the output will equal the magnitude limit.
`Sgn`	Compute the signum of its input. The output is ± 1. Note that 0.0 maps into 1.
`Sin`	Computes the sine of its input, assumed to be an angle in radians.

`Sinc`	Computes the sinc of its input given in radians. The sinc function is defined as sin(x)/x, with value 1.0 when $x = 0$.
`Sqrt`	Computes the square root of its input.

Other Nonlinear Functions

`DB`	Convert input to a decibels (dB) scale. Zero and negative values are assigned the value *min* (default -100). The *inputIsPower* parameter should be set to YES if the input signal is a power measurement (vs. an amplitude measurement).
`PcwzLinear`	This star implements a piecewise linear mapping from the list of (x,y) pairs, which specify the breakpoints in the function. The sequence of x values must be increasing. The function implemented by the star can be represented by drawing straight lines between the (x,y) pairs, in sequence. The default mapping is the 'tent' map, in which inputs between -1.0 and 0.0 are linearly mapped into the range -1.0 to 1.0. Inputs between 0.0 and 1.0 are mapped into the same range, but with the opposite slope, 1.0 to -1.0. If the input is outside the range specified in the "x" values of the breakpoints, then the appropriate extreme value will be used for the output. Thus, for the default map, if the input is -2.0, the output will be -1.0. If the input is +2.0, the output will again be -1.0.
`powerEst`	Estimate the power in decibels (dB) by filtering the square of the input using a first-order filter with the time constant given as a number of sample periods.
`powerEstCx`	Like `powerEst`, but for complex inputs.
`powerEstLin`	Same as `powerEst`, but the output is on a linear scale instead of decibels (dB).
`Table`	This star implements a real-valued lookup table indexed by an integer-valued input. The input must lie between 0 and *N*-1, inclusive, where *N* is the size of the table. The *values* parameter specifies the table. Its first element is indexed by a zero-valued input. An error occurs if the input value is out-of-bounds.
`TableCx`	Table lookup for complex values.
`TableInt`	Table lookup for integer values.
`TclScript`	(two icons) Invoke a Tcl script that can optionally define a procedure that is invoked every time the star fires. That procedure can read the star's inputs and update the value of the outputs.

5.2.5 Logic stars

The logic palette shown in figure 5-7 is made up of only three stars. Each star has multiple icons representing a variety of configurations.

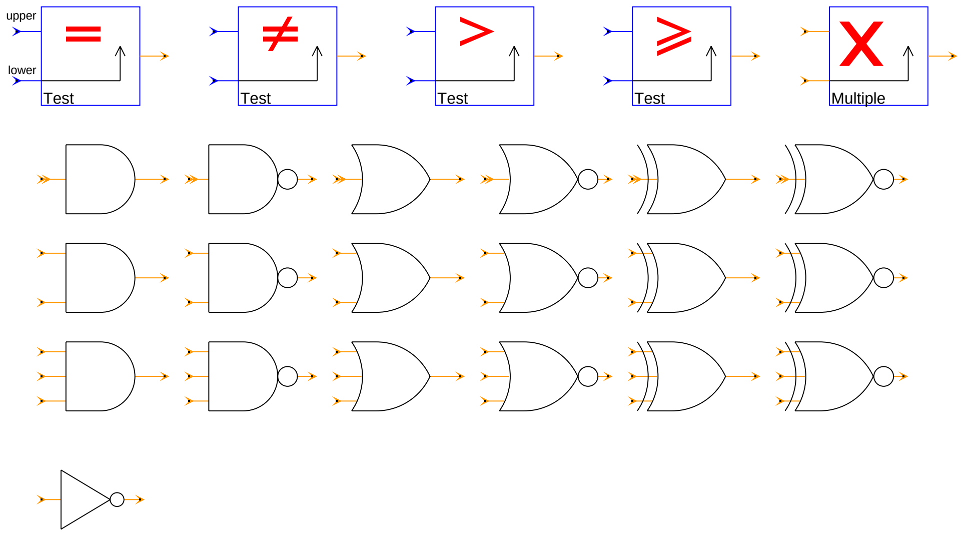


FIGURE 5-7: Logic stars in the SDF palette.

`Test`	(four icons) Compare two inputs. The test condition can be any of {`EQ NE GT GE`} or {`== != > >=`}, resulting in equals, not equals, greater than, or greater than or equals. The four icons represent these possibilities. If *crossingsOnly* is `TRUE`, then the output is non-zero only when the outcome of the test changes from `TRUE` to `FALSE` or `FALSE` to `TRUE`. In this case, the first output is always `TRUE`.
`Multiple`	(one icon) Output a 1 if top input is a multiple of bottom input.
`Logic`	(19 icons) This star applies a logical operation to any number of inputs. The inputs are integers interpreted as Booleans, where zero is a `FALSE` and nonzero is a `TRUE`. The logical operations supported are {`NOT`, `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`}, with any number of inputs.

5.2.6 Control stars

Control stars (figure 5-8) manipulate the flow of tokens. All of these stars are polymorphic; they operate on any data type. From left to right, top to bottom, they are:

Single-Rate Operations

`Fork`	(five icons) Copy input particles to each output. Note that a fork is automatically inserted in a schematic when a single output is sent to more than one input. However, when a delay is needed on one of the connections, then an explicit fork star must be used.
`Reverse`	On each execution, read a block of *N* samples (default 64) and write them out backwards.

`Transpose`	Transpose a rasterized matrix (one that is read as a sequence of particles, row by row, and written in the same form). The number of particles produced and consumed equals the product of *samplesInaRow* and *numberOfRows*.
`TkBreakPt`	A conditional break point. Each time this star executes, it evaluates its conditional expression. If the expression evaluates to true, it causes the run to pause.
`Trainer`	Pass the value of the *train* input to the output for the first *trainLength* samples, then pass the *decision* input to the output. This star is designed for use with adaptive equalizers that require a training sequence at start-up, but it can be used whenever one sequence is used during a start-up phase, and another sequence after that.

Multirate Operations

`Commutator`	(four icons) Synchronously combine N input streams (where N is the number of inputs) into one output stream. The star consumes B input particles from each input (where B is the *blockSize*), and produces $N \times B$ particles on the output. The first B particles on the output come from the first input, the next B particles from the next input, etc.

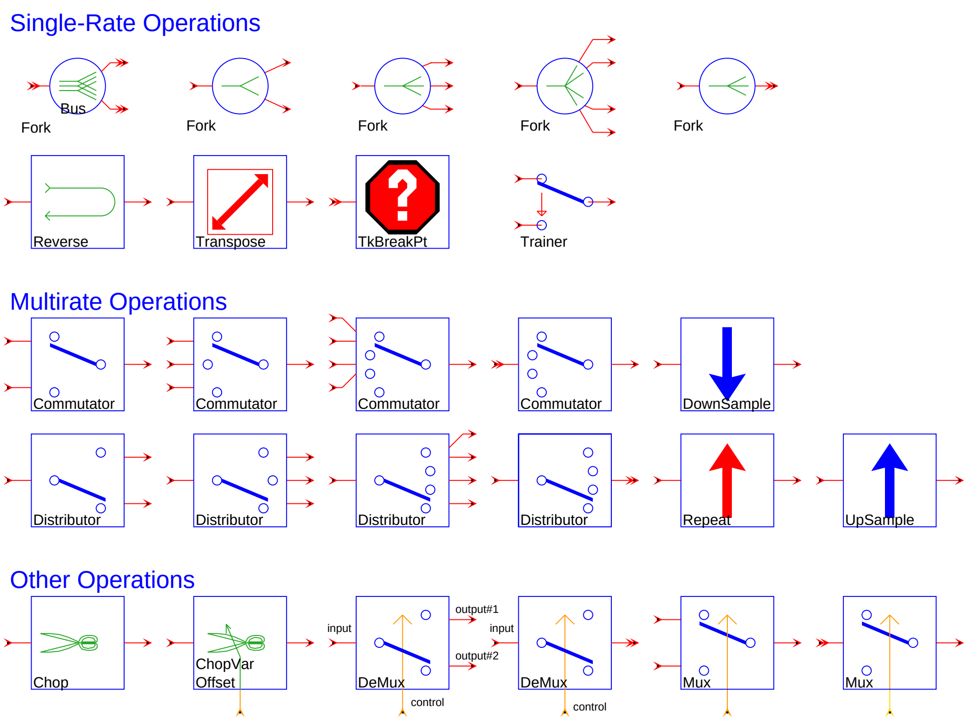


FIGURE 5-8: Control stars for the SDF domain.

`DownSample`	Decimate by a given *factor* (default 2). The *phase* tells which sample of the last *factor* samples to output. If *phase* = 0 (by default), the most recent sample is the output, while if *phase* = *factor* −1 the oldest sample is the output. Note that *phase* has the opposite sense of the *phase* parameter in the `UpSample` star, but the same sense as the *phase* parameter in the `FIR` star.
`Distributor`	(four icons) Synchronously split one input stream into *N* output streams, where *N* is the number of outputs. The star consumes *N* × *B* input particles, where *B* is the *blockSize* parameter, and sends the first *B* particles to the first output, the next *B* particles to the next output, etc.
`Repeat`	Repeat each input sample a specified number of times.
`UpSample`	Upsample by a given factor (default 2), giving inserted samples the value *fill* (default 0.0). The *phase* parameter (default 0) tells where to put the sample in an output block. A *phase* of 0 says to output the input sample first, followed by the inserted samples. The maximum *phase* is equal to *factor* - 1. Although the *fill* parameter is a floating-point number, if the input is of some other type, such as complex, then the *fill* particle will be obtained by casting *fill* to the appropriate type.

Other Operations

`Chop`	On each execution, this star reads a block of *nread* particles and writes them to the output with the given offset. The number of particles written is given by *nwrite*. The output block contains all or part of the input block, depending on *offset* and *nwrite*. The *offset* specifies where in the output block the first (oldest) particle in the input block will lie. If *offset* is positive, then the first *offset* output particles will be either particles consumed on previous firings (if *use_past_inputs* parameter is `YES`), or zero (otherwise). If *offset* is negative, then the first *offset* input particles will be discarded.
`ChopVarOffset`	This star has the same functionality as the `Chop` star except the *offset* parameter is determined at run time by a control input.
`DeMux`	(two icons) Demultiplex one input onto any number of output streams. The star consumes *B* particles from the input, where *B* is the *blockSize*. These *B* particles are copied to exactly one output, determined by the “control” input. The other outputs get a zero of the appropriate type. Integers from 0 through *N* − 1 are accepted at the “control” input, where *N* is the number of outputs. If “control” is outside this range, all outputs get zeros.
`Mux`	(two icons) Multiplex any number of inputs onto one output

stream. *B* particles are consumed on each input, where *B* is the *blockSize*. But only one of these blocks of particles is copied to the output. The one copied is determined by the "control" input. Integers from 0 through $N - 1$ are accepted at the "control" input, where *N* is the number of inputs. If "control" is outside this range, an error is signaled.

5.2.7 Conversion stars

The palette in figure 5-9 shows a collection of stars for format conversions of various types. The first two rows contain stars with functions that are fundamentally different from the automatic type conversion performed by Ptolemy. From left to right, top to bottom, they are:

Complex data type formats

`CxToRect`	Convert a complex input to real and imaginary parts.
`RectToCx`	Convert real and imaginary inputs to a complex output.
`RectToPolar`	Convert real and imaginary inputs into magnitude and phase

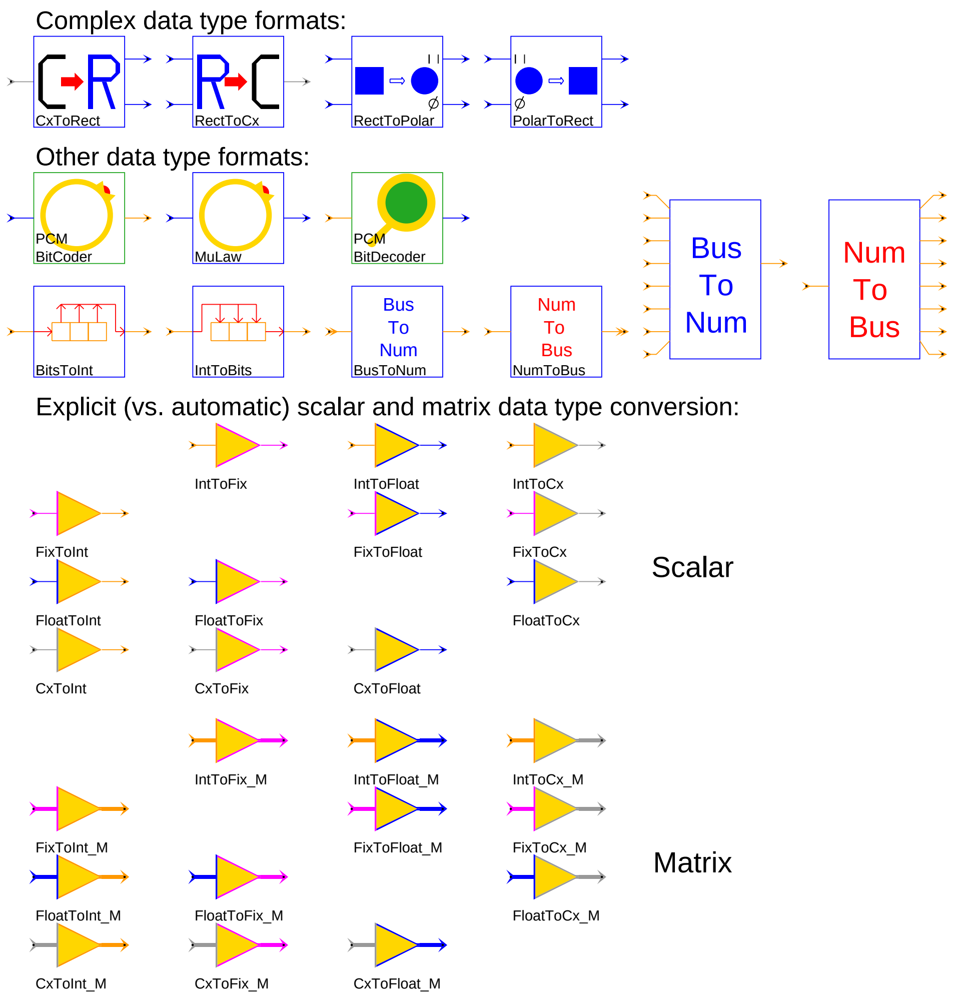


FIGURE 5-9: Type conversion stars for the SDF domain.

	form. The phase output is in the range $-\pi$ to π.
`PolarToRect`	Convert magnitude and phase to rectangular form.

Other data type formats

`PCMBitCoder`	Encode voice samples for a 64 kbps bit stream using CCITT Recommendation G.711. The input is one 8 kHz sample of voice data and the output is the eight-bit codeword (the low-order 8 bits of an integer) representing the quantized samples.
`MuLaw`	This star encodes its input into an 8 bit representation using the nonlinear companding μ-law. It is similar to `PCMBitCoder`, but it does the conversion in a single star, rather than a galaxy.
`PCMBitDecoder`	Decode 8-bit PCM codewords that were encoded using `PCM-BitCoder`.
`BitsToInt`	The integer input sequence is interpreted as a bit stream in which any non-zero value is a "1" bit. This star consumes *nBits* successive bits from the input, packs them into an integer, and outputs the resulting integer. The first received bit becomes the most significant bit of the output. If *nBits* is larger than the integer wordsize, then the first bits received will be lost. If *nBits* is smaller than the wordsize minus one, then the output integer will always be non-negative.
`IntToBits`	Read the least significant *nBits* bits from an integer input, and output the bits as integers serially on the output, most significant bit first.
`BusToNum`	(two icons) This star accepts a number of input bit streams, where this number should not exceed the word size of an integer. Each bit stream has integer particles with values 0, 3, or anything else. These are interpreted as binary 0, tri-state, or 1, respectively. When the star fires, it reads one input bit from each input. If any of the input bits is tri-stated, the output will be the previous output (or the initial value of the *previous* parameter if the firing is the first one). Otherwise, the bits are assembled into an integer word, assuming two's complement encoding, and sign extended. The resulting signed integer is sent to the output. This star is particularly useful for interfacing to digital logic simulation domains.
`NumToBus`	(two icons) This star accepts an integer and outputs the low-order bits that make up the integer on a number of outputs, one bit per output. The number of outputs should not exceed the word size of an integer. This star is particularly useful for interfacing to digital logic simulation domains.

Automatic type conversion, as implemented in Ptolemy 0.7, has limitations. If a given output port has more than one destination, then all destinations must have the same type input. This is true even if an explicit `fork` star is used. Explicit type conversions are needed to get around this limitation. For this reason, the palette in figure 5-9 also contains a set of type conversions that behave exactly the same way the automatic type conversions behave.

`IntToFix`	Convert an integer input to a fixed-point output.
`IntToFloat`	Convert an integer input to a floating-point output.
`IntToCx`	Convert an integer input to a complex output.
`FixToInt`	Convert a fixed-point input to an integer output.
`FixToFloat`	Convert a fixed-point input to a floating-point output.
`FixToCx`	Convert a fixed-point input to a complex output.
`FloatToInt`	Convert a floating-point input to an integer output.
`FloatToFix`	Convert a floating-point input to a fixed-point output.
`FloatToCx`	Convert a floating-point input to a complex output.
`CxToInt`	Convert a complex input to an integer output.
`CxToFix`	Convert a complex input to a fixed-point output.
`CxToFloat`	Convert a complex input to a floating-point output.

Matrix Conversion Stars

The following type conversions construct a new matrix of the destination type by converting each element of the old matrix as it is copied to the new one. For `FixMatrix` types, the precision is specified as a parameter of the conversion star. The actual conversions are implemented using the cast conversion in the underlying class, except for the conversions to the `FixMatrix` type which are more complex because they involve possible changes in precision and require a rounding option. The stars provided are:

`IntToFix_M`	Convert an integer input matrix to a fixed-point output matrix.
`IntToFloat_M`	Convert an integer input matrix to a floating-point output matrix.
`IntToCx_M`	Convert an integer input matrix to a complex output matrix.
`FixToInt_M`	Convert a fixed-point input matrix to an integer output matrix.
`FixToFloat_M`	Convert a fixed-point input matrix to a floating-point output matrix.
`FixToCx_M`	Convert a fixed-point input matrix to a complex output matrix.
`FloatToInt_M`	Convert a floating-point input matrix to an integer output matrix.
`FloatToFix_M`	Convert a floating-point input matrix to a fixed-point output matrix.
`FloatToCx_M`	Convert a floating-point input matrix to a complex output

	matrix.
`CxToInt_M`	Convert a complex input matrix to an integer output matrix.
`CxToFix_M`	Convert a complex input matrix to a fixed-point output matrix.
`CxToFloat_M`	Convert a complex input matrix to a floating-point output matrix.

5.2.8 Matrix stars

The stars in the matrix palette (figure 5-10) operate on particles that represent matrices with floating-point, fixed-point, complex, or integer entries. Most of the work is done in the underlying matrix classes, `FloatMatrix`, `ComplexMatrix`, `FixMatrix`, and `IntMatrix`. These classes are treated as ordinary particles. In Pigi, matrix types are indicated with thick terminal stems, where the color of the terminal stem corresponds to the data type of the matrix elements.

The Matrix conversion stars are in the conversion palette, see “Matrix Conversion Stars” on page 5-22 for more information.

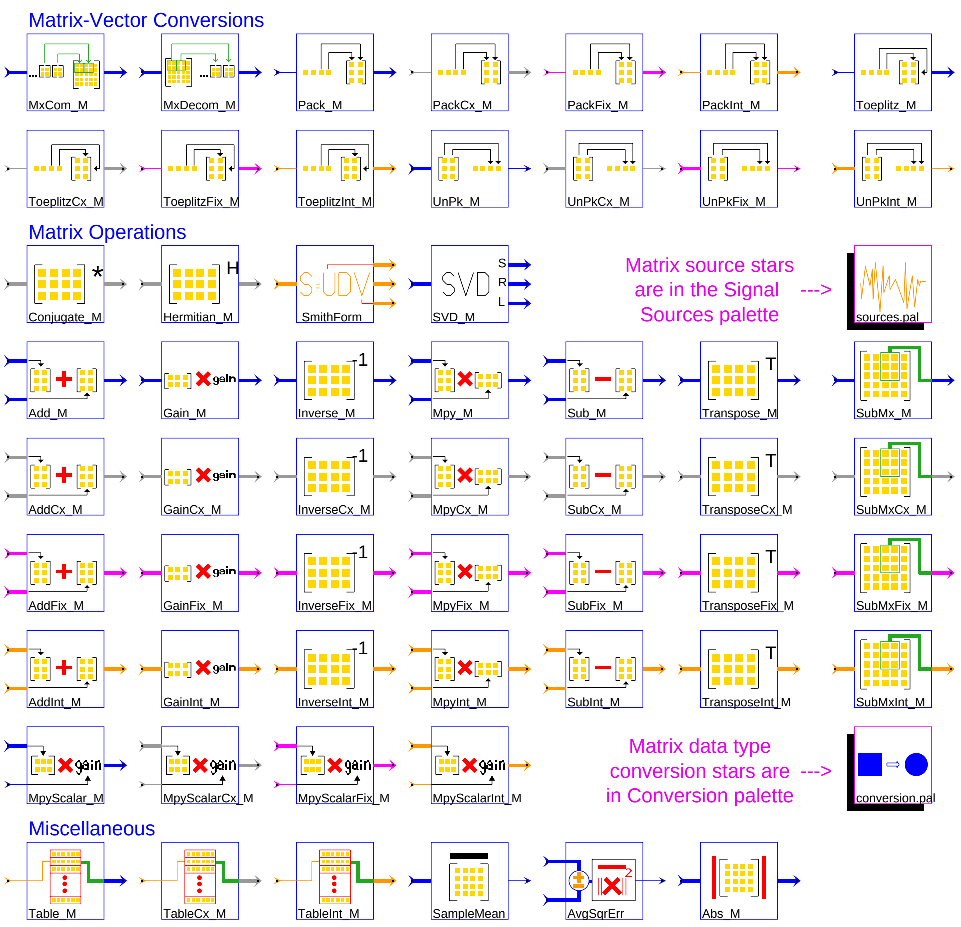


FIGURE 5-10: The matrix palette in the SDF domain. These stars operate on matrices encapsulated in a particles.

Matrix-Vector Conversion

`MxCom_M`	Accept input matrices and create a matrix output. Each input matrix represents a decomposed submatrix of output matrix in row by row. Note that for one output image, we will need a total (*numRows* / *numRowsSubMx*) × (*numCols* / *numColsSubMx*) input matrices.
`MxDecom_M`	Decompose a portion of input matrix into a sequence of submatrices. The desired portion of input matrix is specified by the parameters *startRow*, *startCol*, *numRows*, and *numCols*. Then output each submatrix with dimension *numRowsSubMx* × *numColsSubMx* in row by row. Note that for one input matrix, there will be a total of (*numRows* / *numRowsSubMx*) × (*numCols* / *numColsSubMx*) output matrices.

The following conversions perform more interesting functions. They also come in four versions, one for each data type, and again we only list the floating-point version.

`Pack_M` (4 icons) Produce a matrix with floating-point entries constructed from floating-point input particles. The inputs are put in the matrix in rasterized order, e.g. for a $M \times N$ matrix, the first row is filled from left to right using the first N input particles.

`Toeplitz_M` (4 icons) Generate a floating-point data matrix X, with dimensions (*numRows*,*numCols*), from a stream of *numRows* + *numCols* – 1 input particles organized as shown below:

$$X = \begin{bmatrix} x(M-1) & x(M-2) & \dots & x(0) \\ x(M) & x(M-1) & \dots & x(1) \\ \dots & \dots & \dots & \dots \\ x(N-1) & x(N-2) & \dots & x(N-M) \end{bmatrix}$$

Here *numRows* = $N - M + 1$ and *numCols* = M. This Toeplitz matrix is the form of the matrix that is required by the `SVD_M` star, among others.

`UnPk_M` (4 icons) Read a floating-point matrix and output its elements, row by row, as a stream of floating-point particles.

Matrix operations

The following blocks are functions defined only for the `ComplexMatrix` data type.

`Conjugate_M`	Conjugate a matrix.
`Hermitian_M`	Perform a Hermitian transpose (conjugate transpose) on the input matrix.

The following blocks also appear in the signal processing palette.

`SmithForm` Decompose an integer matrix S into one of its Smith forms S =

	UDV, where *U*, *D*, and *V* are simpler integer matrices. The Smith form decomposition for integer matrices is analogous to singular value decomposition for floating-point matrices.
`SVD_M`	Compute the singular-value decomposition of a Toeplitz data matrix *A* by decomposing *A* into *A* = *UWV'*, where *U* and *V* are orthogonal matrices, and *V'* represents the transpose of *V*. *W* is a diagonal matrix composed of the singular values of *A*, and the columns of *U* and *V* are the left and right singular vectors of *A*.

See "Matrix Sources" on page 5-8 for the Matrix source stars.

The following are usual matrix operations. They are arranged row by row, with one row for each data type (floating point, complex, fixed point, and integer). We list below only the floating point data type, from left to right.

`Add_M`	Add two floating-point matrices.
`Gain_M`	Multiply a floating-point matrix by a static scalar gain value.
`Inverse_M`	Invert a square floating-point matrix.
`Mpy_M`	Multiply two floating-point matrices *A* and *B* to produce matrix *C*. Matrix *A* has dimensions (*numRows*,*X*). Matrix *B* has dimensions (*X*,*numCols*). Matrix *C* has dimensions (*numRows*,*numCols*). The user need only specify *numRows* and *numCols*. An error will be generated if the number of columns in *A* does not match the number of rows in *B*.
`Sub_M`	Subtract floating-point matrix *B* from *A*.
`Transpose_M`	Transpose a floating-point matrix read as a single particle.
`SubMx_M`	Find a submatrix of the input matrix.
`MpyScalar_M`	Multiply a floating-point matrix by a scalar gain value given in parameter.

Miscellaneous

`Table_M`	(3 stars for floating-point, complex and integer) This star implements a lookup table indexed by an integer-valued input. The output is a matrix. The input must lie between 0 and $N - 1$, inclusive, where *N* is the number of matrices in the table. The *floatTable* parameter specifies the entries of matrices in the table. Note that the entries of each matrix in the table should be given in row major ordering. The first matrix in the table is indexed by a zero-valued input. An error occurs if the input value is out of bounds.
`SampleMean`	Find the average amplitude of the components of the input matrix.
`AvgSqrErr`	Find the average squared error between two input sequences of

	matrices.
`Abs_M`	Return the absolute value of each entry of the floating-point matrix.

5.2.9 Matlab stars

The Matlab stars provide an interface between Ptolemy and Matlab, a numeric computation and visualization environment from The Math Works, Inc. Each Matlab star can contain a single Matlab function, command, statement, or several statements. Ptolemy handles the conversion of inputs into Matlab format and the results from Matlab into Ptolemy format. For the Matlab stars to work, Matlab version 4.1 or later must be installed. Matlab is not distributed with Ptolemy[1]. If a Matlab star is run and Matlab is not installed, then Ptolemy will report an error. All Matlab stars send their commands to the same Matlab process.

Xavier Warzee of Thomson-CSF provided a method of running Matlab on a remote machine and obtaining the results from within Ptolemy. If a simulation needs to start Matlab, then the `PTMATLAB_REMOTE_HOST` environment variable is checked. If this variable is set, then its value is assumed to be the name of the remote machine to run Matlab on. The remote Matlab process is started up with the Unix `rsh` command. Once the remote process is running, if the `MATLAB_SCRIPT_DIR` environment variable is set, then its value is passed to the remote Matlab process as part of the command

```
    path(path.'MATLAB_SCRIPT_DIR')
```

where *`MATLAB_SCRIPT_DIR`* is the value of that variable on the local machine.

Internally, Matlab distinguishes between real matrices and complex matrices. As a

1. Contact The Math Works, Inc., Cochituate Place, 24 Prime Park Way, Natick, Mass. 01760-1500, USA, Phone: (508) 653-1415. Their Web site is http://www.mathworks.com/.

consequence, in Figure 5-11 there are two types of Matlab stars: one outputs floating-point

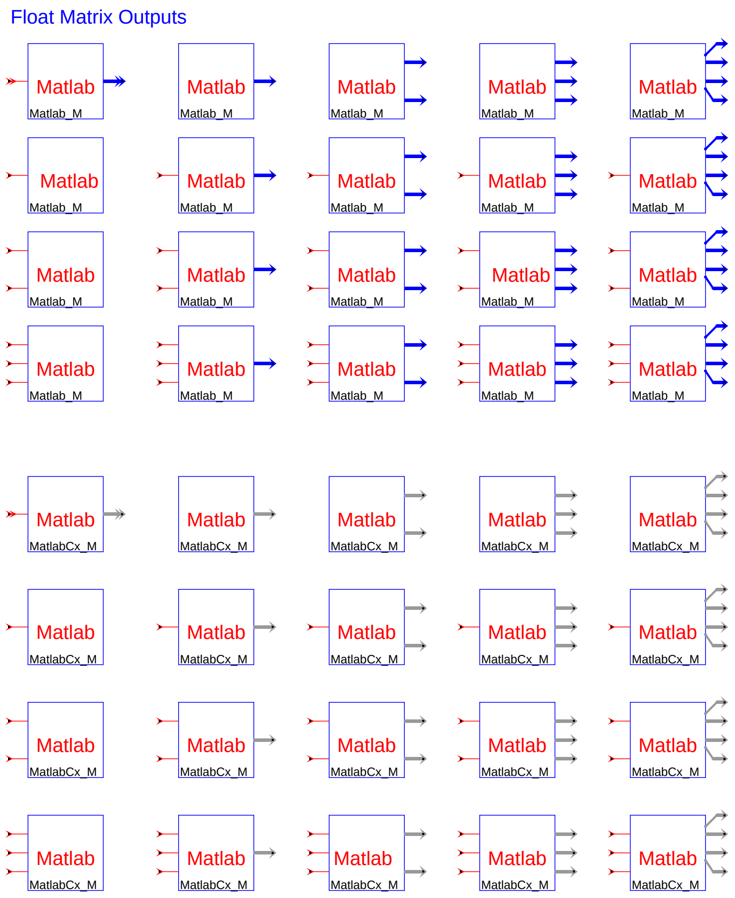


FIGURE 5-11: Matlab stars in the SDF domain.

matrices and one outputs complex-valued matrices. These stars can take any number of inputs provided that the inputs have the same data type (floating point or complex). The two types of Matlab stars are:

`Matlab_M`	Evaluate a Matlab expression and output the result as floating-point matrices.
`MatlabCx_M`	Evaluate a Matlab expression and output the result as complex-valued matrices.

The implementation of Matlab stars is built on Matlab's engine interface. The interface is managed by a base star, `SDFMatlab`. The base star does not have any inputs or outputs. It provides methods for starting and killing a Matlab process, evaluating Matlab commands, managing Matlab figures, changing directories in Matlab, and passing Ptolemy matrices in and out of Matlab. Currently, the base star does support real- and complex-valued matrices, but not Matlab's other two matrix data types, sparse and string matrices.

Figures generated by a Matlab star are managed according to the value of the star's *DeleteOldFigures* parameter. If `TRUE` or `YES`, then the Matlab star will close any plots, graphics, etc., that it has generated when the Matlab star is destroyed (e.g., when the run panel in the graphical interface is closed). Otherwise, the figures remain until Ptolemy exits. For standalone programs generated by compile-SDF, it is better to set this parameter to `NO` so that the plots will not disappear when then standalone programs finishes.

There are several ways in which Matlab commands can be specified in the Matlab stars. The Matlab stars `Matlab_M` and `MatlabCx_M` have a parameter *MatlabFunction*. If only a Matlab function name is given for this parameter, then the function is applied to the inputs in the order they are numbered and the output(s) of the function is (are) sent to the star's outputs. For example, specifying `eig` means to perform the eigendecomposition of the input. The function will be called to produce one or two outputs, according to how many output ports there are. If there is a mismatch in the number of inputs and/or outputs between the Ptolemy star and the Matlab function, Ptolemy will report the error generated by Matlab.

The user may also specify how the inputs are to be passed to a Matlab function or how the outputs are taken from the Matlab function. For example, consider a two-input, two-output Matlab star to perform a generalized eigendecomposition. The command

```
[output#2, output#1] = eig( input#2, input#1 )
```

says to perform the generalized eigendecomposition on the two input matrices, place the generalized eigenvectors on output#2, and the eigenvalues (as a diagonal matrix) on output#1. Before this command is sent to Matlab, the pound characters '#' are replaced with underscore '_' characters because the pound character is illegal in a Matlab variable name.

The Matlab stars also allow a sequence of commands to be evaluated. Continuing with the previous example, we can plot the eigenvalues on a graph after taking the generalized eigendecomposition:

```
[output#2, output#1] = eig( input#2, input#1 );
plot( output#1 )
```

When entering such a collection of commands in Ptolemy, both commands would appear on the same line without a newline after the semicolon. In this way, very complicated Matlab commands can be built up. We can make the plot of eigenvalues always appear in the same plot without interfering with other plots generated by other Matlab stars:

```
[output#2, output#1] = eig( input#2, input#1 );
if ( exist('myEigFig') == 0 ) myEigFig = figure; end;
figure(myEigFig);
plot( output#1);
```

For more information about using Matlab stars, please refer to the Matlab demonstrations.

5.2.10 UltraSparc Native DSP

The Visual Instruction Set (VIS) demos only run on Sun Ultrasparc workstations with the Sun unbundled CC compiler and a Ptolemy tree that has been compiled with the `PTARCH` variable set to `sol2.5.cfront`. The VIS demos will not work the Gnu compilers. You must have the Sun Visual Instruction Set Development kit installed, see `http://www.sun.com/sparc/vis/vsdkfaq.html`.

The palette shown in figure 5-12 has icons for the library of Sun UltraSparc Visual

Instruction Set (VIS) stars.

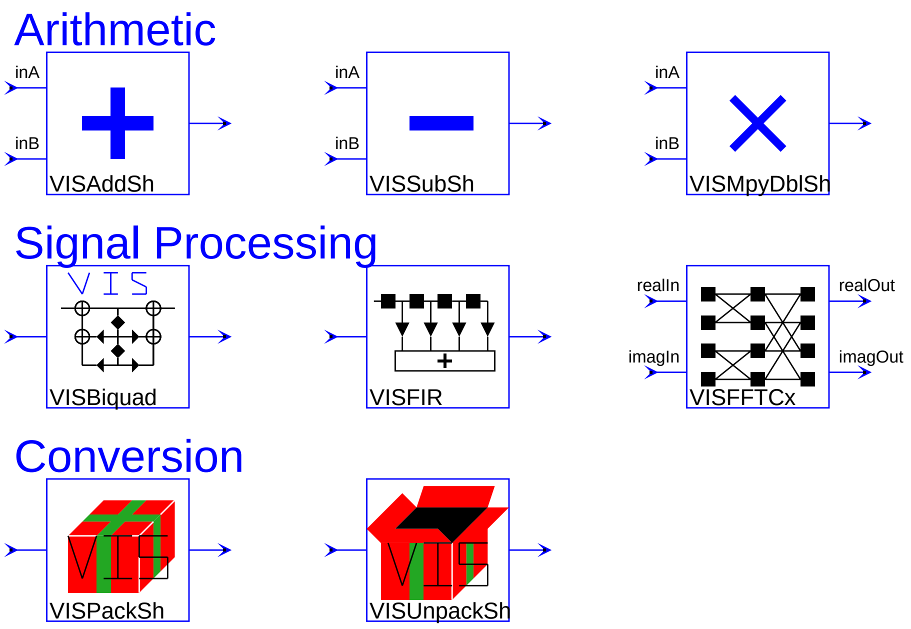


FIGURE 5-12: Sun UltraSparc Visual Instruction Set (VIS) DSP stars in the SDF domain.

`VISAddSh`	Add the shorts in a 16 bit partitioned float to the corresponding shorts in a 16 bit partitioned float. The result is four signed shorts that is returned as a single floating point number. There is no saturation arithmetic so that overflow results in wraparound.
`VISSubSh`	Subtract the shorts in a 16 bit partitioned float to the corresponding shorts in a 16 bit partitioned float.The result is four signed shorts that is returned as a single floating point number. There is no saturation arithmetic so that overflow results in wraparound.
`VISMpyDblSh`	Multiplies the shorts in a 16 bit partitioned float to the corresponding shorts in a 16 bit partitioned float. The result is four signed shorts that is returned as a single floating point number. Each multiplication results in a 32 bit result, which is then rounded to 16 bits.
`VISBiquad`	An IIR Biquad filter.
`VISFIR`	A finite impulse response (FIR) filter.
`VISFFTCx`	A single complex sequence FFT using radix 2.
`VISPackSh`	Pack four floating point numbers into a single floating point number.

`VISUnPackSh`	Unpack a single floating point number into four floating point numbers.

5.2.11 Signal processing stars

The palette shown in figure 5-13 has icons for the library of signal processing functions. Simple time-domain filtering operations come first.

Filters

`Biquad` A two-pole, two-zero Infinite Impulse Response filter (a biquad). The default is a Butterworth filter with a cutoff at 0.1 times the sample frequency. The transfer function is

$$H(z) = \frac{n_0 + n_1 z^{-1} + n_2 z^{-2}}{1 + d_1 z^{-1} + d_2 z^{-2}} .$$

`Convolve` Convolve two causal finite sequences of floating point numbers. The *truncationDepth* parameter specifies the number of terms used in the convolution sum. Set *truncationDepth* larger than

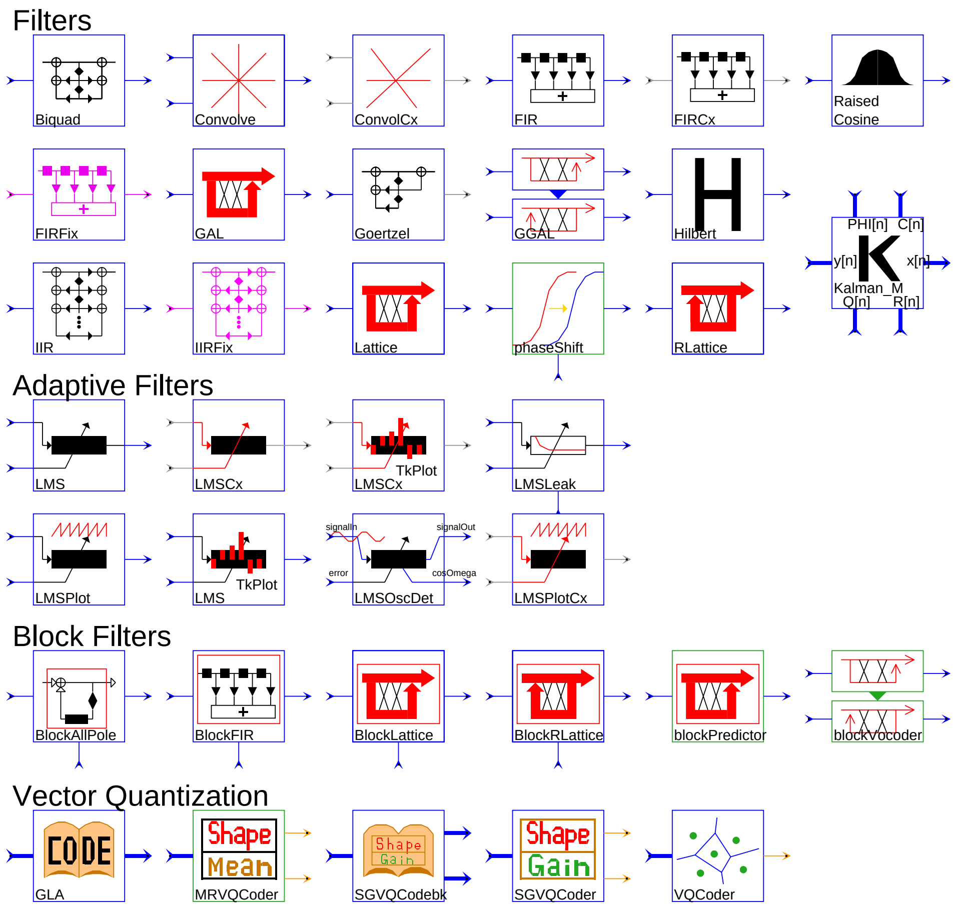


FIGURE 5-13: The signal processing (dsp) palette of the SDF domain.

	the number of output samples of interest.
`ConvolveCx`	Convolve two causal finite sequences of complex numbers. The *truncationDepth* parameter specifies the number of terms used in the convolution sum. Set *truncationDepth* larger than the number of output samples of interest.
`FIR`	A Finite Impulse Response (FIR) filter. Coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear-phase, lowpass filter. To read coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path. Rational sampling rate changes, implemented by polyphase multirate filters, is also supported.
`FIRCx`	A complex FIR filter. Coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear phase, lowpass filter. To read coefficients from a file, use the syntax: `< fileName`, preferably specifying a complete path. Real and imaginary parts should be paired with parentheses, e.g. (1.0, 0.0). Polyphase multirate filtering is also supported.
`RaisedCosine`	An FIR filter with a magnitude frequency response that is shaped like the standard raised cosine or square-root raised cosine used in digital communications. By default, the star upsamples by a factor of 16, so 16 outputs will be produced for each input unless the *interpolation* parameter is changed.
`FIRFix`	An FIR filter with fixed-point capabilities. The fixed-point coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear phase lowpass filter. To read coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path. Polyphase multirate filtering is also supported.
`Kalman_M`	Output the state vector estimates of a Kalman filter using a one-step prediction algorithm.
`GAL`	A Gradient Adaptive Lattice filter.
`Goertzel`	Second-order recursive computation of the kth coefficient of an N-point DFT using Goertzel's algorithm.
`GGAL`	Ganged Gradient Adaptive Lattice filters.
`Hilbert`	Output the (approximate) Hilbert transform of the input signal. This star approximates the Hilbert transform by using an FIR filter, and is derived from the `FIR` star.
`IIR`	An Infinite Impulse Response (IIR) filter implemented in direct form II. The transfer function is of the form

$$H(z) = G\frac{N(1/z)}{D(1/z)} ,$$

where *N*() and *D*() are polynomials. The parameter *gain* specifies *G*, and the floating-point arrays *numerator* and *denominator* specify *N*() and *D*(), respectively. Both arrays start with the constant terms of the polynomial and decrease in powers of *z* (increase in powers of 1/*z*). Note that the constant term of *D* is not omitted, as is common in other programs that assume it is always normalized to unity.

`IIRFix`	This is a fixed-point version of the `IIR` star. The coefficient precision, input precision, accumulation precision, and output precision can all be separately specified.
`Lattice`	An FIR lattice filter. The default reflection coefficients form the optimal predictor for a particular 4th-order AR random process. To read other reflection coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path.
`phaseShift`	This galaxy applies a phase shift to a signal according to the "shift" input. If the "shift" input value is time varying, then its slope determines the instantaneous frequency shift.
`RLattice`	A recursive (IIR) lattice filter. The default coefficients implement the synthesis filter for a particular 4th-order AR random process. To read reflection coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path.

Adaptive Filters

`LMS`	An adaptive filter using the Least-Mean Square (LMS) adaptation algorithm. The initial coefficients are given by the *taps* parameter. The default initial coefficients give an 8th order, linear phase lowpass filter. To read default coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path. This star, which is derived from `FIR`, supports decimation, but not interpolation.
`LMSCx`	Complex version of the `LMS` star.
`LMSCxTkPlot`	This star is just like the `LMSCx` star, but with an animated Tk display of the taps, plus associated controls.
`LMSLeak`	An LMS adaptive filter in which the step size is input (to the "step" input) every iteration. In addition, the *mu* parameter specifies a leakage factor in the updates of the filter coefficients.
`LMSPlot`	This star is just like the `LMS` star, except that, in addition to the functions of `LMS`, it makes a plot of the tap coefficients. It can produce two types of plots: a plot of the final tap values or a plot that traces the time evolution of each tap value. The time evolu-

	tion is obtained if the value of the parameter *trace* is `YES`.
`LMSTkPlot`	This star is just like the `LMS` star, but with an animated Tk display of the taps, plus associated controls.
`LMSOscDet`	This filter tries to lock onto the strongest sinusoidal component in the input signal, and outputs the current estimate of the cosine of the frequency of the strongest component and the error signal. It is a three-tap LMS filter whose first and third coefficients are fixed at one. The second coefficient is adapted. It is a normalized version of the Direct Adaptive Frequency Estimation Technique.
`LMSPlotCx`	Complex version of `LMSPlot`. Separate plots are generated for the magnitude and phase of the filter coefficients.

Block Filters

The next group of stars perform “block filtering”, which means that on each firing, they read a set of input particles all at once, process them, and produce a set of output particles. The number of particles in a set is specified by the *blockSize* parameter.

`BlockAllPole` This star implements an all pole filter with the denominator coefficients of the transfer function externally supplied. For each set of coefficients, a block of input samples is processed, all in one firing. The transfer function is

$$H(z) = \frac{1}{1 - D(z)}$$

where the coefficients of $D(z)$ are externally supplied.

`BlockFIR`	This star implements an FIR filter with coefficients that are periodically updated from the outside. For each set of coefficients, a block of input samples is processed, all in one firing.
`BlockLattice`	A block forward lattice filter. It is identical to the `Lattice` star except that the reflection coefficients are updated each time the star fires by reading the “coefs” input. The *order* parameter indicates how many coefficient should be read. The *blockSize* parameter specifies how many data samples should be processed for each set of coefficients.
`BlockRLattice`	A block recursive (IIR) lattice filter. It is identical to the `RLattice` star, except that the reflection coefficients are updated each time the star fires by reading the “coefs” input. The *order* and *blockSize* parameters have the same interpretation as in the `BlockLattice` star.
`blockPredictor`	A block predictor galaxy used in speech processing.
`blockVocoder`	A block vocoder galaxy.

Vector Quantization

Quantization is the heart of converting analog signals to digital signals. Traditional techniques are based on *scalar* coding which quantizes symbols, such as pixels in images, one by one. On the other hand, vector quantization can perform better by operating the quantization on groups of symbols instead of individual symbols.

`GLA`	Use the Generalized Lloyd Algorithm (GLA) to yield a codebook from input training vectors. Note that each input matrix will be viewed as a row vector in row by row. Each row of output matrix represents a codeword of the codebook.
`MRVQCoder`	Mean removed vector quantization coder.
`SGVQCodebk`	Jointly optimized codebook design for shape-gain vector quantization. Note that each input matrix will be viewed as a row vector in row by row. Each row of first output matrix represents a codeword of the shape codebook. Each element of the second output matrix represents a codeword of the gain codebook.
`SGVQCoder`	Shape-gain vector quantization encoder. Note that each input matrix will be viewed as a row vector in row by row.
`VQCoder`	Full search vector quantization encoder. It consists in finding the index of the nearest neighbor in the given codebook corresponding to the input matrix. Note that each input matrix will first be viewed as a row vector in row by row, in order to find the nearest neighbor codeword in the codebook.

5.2.12 Spectral analysis

The group of stars shown in figure 5-14 are concerned with various signal analysis algorithms.

`autocorrelation`	Estimate a power spectrum using the autocorrelation method, a method that uses the Levinson-Durbin algorithm to compute linear predictor coefficients, and then uses these coefficients to construct an approximate maximum entropy power spectrum estimate.
`blockFFT`	An overlap and add implementation of the FFT.
`burg`	Estimate a power spectrum using Burg’s method, a method that computes linear predictor coefficients, and then uses them to construct a maximum entropy power spectrum estimate.
`Burg`	This star uses Burg's algorithm to estimate the linear predictor coefficients of an input random process. These coefficients are produced both in autoregressive form (on the “lp” output) and in lattice filter form (on the “refl” output). The “errPower” output is the power of the prediction error as a function of the predictor order. This star is used in the `burg` galaxy.
`DB`	Convert input to a decibel (dB) scale. Zero and negative values

	are assigned the value *min* (default -100). The *inputIsPower* parameter should be set to `YES` if the input signal is a power measurement (vs. an amplitude measurement).
`DTFT`	Compute the discrete-time Fourier transform (DTFT) at frequency points specified on the “omega” input.
`FFTCx`	Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter *order* (default 8) is the log base 2 of the transform size. The parameter *size* (default 256) is the number of samples read (<= 2^*order*). The parameter *direction* (default 1) is 1 for the forward, -1 for the inverse FFT.
`GoertzelPower`	Second-order recursive computation of the power of the kth coefficient of an N-point DFT using Goertzel’s algorithm. This form is used in touch-tone decoding.
`LevDur`	This star uses the Levinson-Durbin algorithm to compute the linear predictor coefficients of a random process, given its autocorrelation function as an input. These coefficients are produced both in autoregressive form (on the “lp” output) and in lattice filter form (on the “refl” output). The “errPower” output is the power of the prediction error as a function of the predictor order.

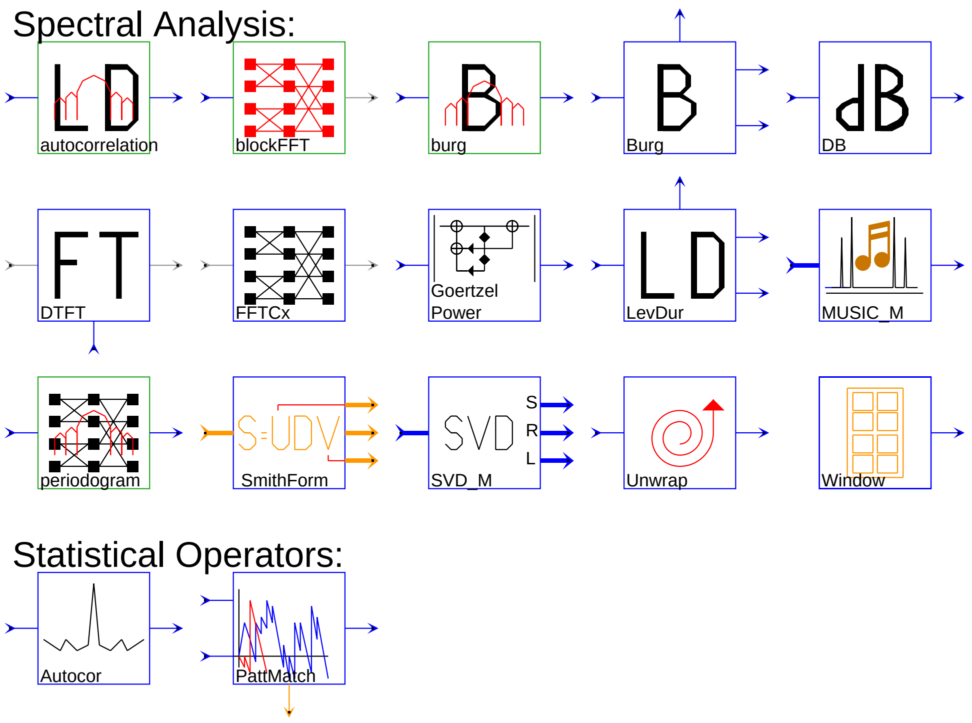


FIGURE 5-14: The spectral analysis palette of the SDF domain

`MUSIC_M`	This star is used to estimate the frequencies of some specified number of sinusoids in a signal. The output is the eigenspectrum of a signal, such that the locations of the peaks of the eigenspectrum correspond to the frequencies of the sinusoids in the signal. The input is the right singular vectors in the form generated by the `SVD_M` star. The MUSIC algorithm (multiple signal characterization) is used.
`periodogram`	Estimate a power spectrum using the periodogram method. This consists in computing the magnitude squared of the DFT of a set of observations of the signal. The FFT algorithm is used.
`SmithForm`	Decompose an integer matrix *S* into one of its Smith forms *S* = *UDV*, where *U*, *D*, and *V* are simpler integer matrices. The Smith form decomposition for integer matrices is analogous to singular value decomposition for floating-point matrices.
`SVD_M`	Compute the singular-value decomposition of a Toeplitz data matrix *A* by decomposing *A* into *A* = *UWV'*, where *U* and *V* are orthogonal matrices, and *V'* represents the transpose of *V*. *W* is a diagonal matrix composed of the singular values of *A*, and the columns of *U* and *V* are the left and right singular vectors of *A*.
`Unwrap`	Unwraps a phase plot, removing discontinuities of magnitude 2π. This star assumes that the phase never changes by more than π in one sample period. It also assumes that the input is in the range $[-\pi,\pi]$.
`Window`	Generate standard window functions or periodic repetitions of standard window functions. The possible functions are `Rectangle`, `Bartlett`, `Hanning`, `Hamming`, `Blackman`, `SteepBlackman`, and `Kaiser`. One period of samples is produced on each firing. This star is also found in the signal sources palette.

Miscellaneous signal processing blocks

`Autocor`	Estimate an autocorrelation function by averaging input samples. Both biased and unbiased estimates are supported.
`PattMatch`	This star accepts a template and a search window. The template is slid over the window one sample at a time, and cross correlations are calculated at each step. The cross-correlations are output on the “values” output. The “index” output is the value of the time-shift which gives the largest cross correlation. This index refers to a position on the search window beginning with 0 corresponding to the earliest arrived sample of the search window that is part of the best match with the template.

5.2.13 Communication stars

The limited set of communication stars that have been developed are shown in figure

5-15, and summarized below. Many of these are galaxies, and should be viewed as examples of systems that a user can create.

Sources and pulse shapers

`bits`	Produce “0” with probability *probOfZero*, else produce “1”.
`cosine.pal`	Produce a cosine waveform whose energy is normalized with respect to *Amplitude*. It is used in simulations for binary frequency shift keying (BFSK) demonstrations. This galaxy differs from the cosine star which computes the cosine of the input signal (see “Nonlinear stars” on page 5-13 for more information on the cosine star).
`Hilbert`	Output the approximate Hilbert transform of the input signal. This star approximates the Hilbert transform by using an FIR filter, and is derived from the `FIR` star. The Hilbert star is also in the signal processing palette, which is discussed on page 5-30.
`RaisedCosine`	An FIR filter with a magnitude frequency response shaped like the standard raised cosine or square-root raised cosine used in digital communication. By default, the star upsamples by a factor of 16, so 16 outputs will be produced for each input unless the *interpolation* parameter is changed.

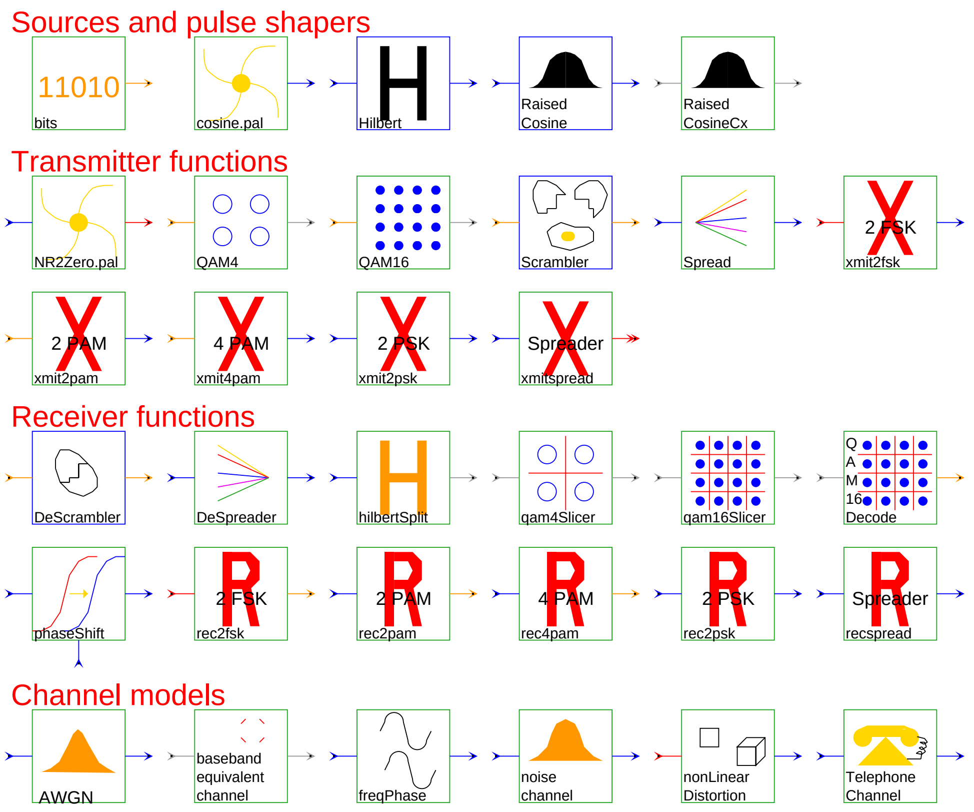


FIGURE 5-15: Communication stars in the SDF domain.

`RaisedCosineCx`	This galaxy uses the `RaisedCosine` star to implement an FIR filter for complex inputs with a raised cosine or square-root raised cosine transfer function.

Transmitter functions

`NR2Zero`	Binary to Nonreturn-to-Zero Signaling Converter
`QAM4`	Encode an input bit stream into a 4-QAM (or 4-PSK) complex symbol sequence.
`QAM16`	Encode an input bit stream into a 16-QAM complex symbol sequence.
`Scrambler`	Scramble the input bit sequence using a feedback shift register. The taps of the feedback shift register are given by the *polynomial* parameter, which should be a positive integer. The n-th bit of this integer indicates whether the n-th tap of the delay line is fed back. The low-order bit is called the 0-th bit, and should always be set. The next low-order bit indicates whether the output of the first delay should be fed back, etc. The default *polynomial* is an octal number defining the V.22bis scrambler.
`Spread`	Frame synchronized direct-sequence spreader.
`xmit2fsk`	Binary frequency shift keying (BFSK) transmitter.
`xmit2pam`	Simple 2-level pulse amplitude modulation (PAM) transmitter.
`xmit4pam`	Simple 4-level pulse amplitude modulation (PAM) transmitter.
`xmit2psk`	Binary 2-level phase shift keying (BPSK) Modulator.
`xmitspread`	Direct-sequence spreader (i.e., spread-spectrum transmitter).

Receiver functions

`DeScrambler`	Descramble the input bit sequence using a feedback shift register. The taps of the feedback shift register are given by the *polynomial* parameter. This is a self-synchronizing descrambler that will exactly reverse the operation of the `Scrambler` star if the polynomials are the same. The low-order bit of the polynomial should always be set.
`DeSpreader`	Frame synchronized direct-sequence despreader.
`hilbertSplit`	This galaxy implements a phase splitter, in which the real-valued input signal is converted to an (approximate) analytic signal. The signal is filtered by the Hilbert block to generate the imaginary part of the output, while the real part is obtained by creating a matching delay.
`qam4Slicer`	This galaxy implements a slicer (decision device) for a 4-QAM (or equivalently, 4-PSK) signal. The output decision is a com-

	plex number with +1 or -1 for each of the real or imaginary parts.
`qam16Slicer`	This galaxy implements a slicer (decision device) for a 16-QAM complex signal. The output decision is a complex number with +1, -1, +3, or -3 for each of the real or imaginary parts.
`qam16Decode`	A 16-QAM decoder similar to the CCITT V22.bis standard. The quadrant is differentially de-encoded.
`phaseShift`	Shifts the phase of the input signal on the *in* input by the shift value on the *shift* input. The phase shifting is implemented by filtering the input signal with a complex FIR filter to convert it into an analytic signal and the complex result is modulated by a complex exponential. If the *shift* value is time varying, then its slope determines the instantaneous frequency shift.
`rec2fsk`	Binary frequency shift keying (BFSK) Receiver.
`rec2pam`	Simple 2-level pulse amplitude modulation (PAM) receiver.
`rec4pam`	Simple 4-level pulse amplitude modulation (PAM) receiver.
`rec2psk`	Binary pulse shift keying (BPSK) Demodulator.
`recspread`	Direct sequence receiver.

Channel models

`AWGNchannel`	Model an additive Gaussian white noise channel with optional linear distortion.
`basebandEquivChannel`	Baseband equivalent channel.
`freqPhase`	Impose frequency offset and/or phase jitter on a signal in order to model channels, such as telephone channels, that suffer these impairments.
`noiseChannel`	A simple channel model with additive Gaussian white noise.
`nonLinearDistortion`	Generate second and third harmonic distortion by squaring and cubing the signal, and adding the results in controlled proportion to the original signal.
`telephoneChannel`	Simulate impairments commonly found on a telephone channel, including additive Gaussian noise, linear and nonlinear distortion, frequency offset, and phase jitter.

5.2.14 Telecommunications

The telecommunications stars are in figure 5-16.

Conversion, Signal Sources, and Signal Tests

`MuLaw`	Transform the input using a logarithmic mapping if the *compress* parameter is true. In telephony, applying the μ-law to eight-bit sampled data is called companding, and it is used to quantize the dynamic range of speech more accurately. The transformation is defined in terms of the non-negative integer parameter *mu*: $\text{output} = \log(1 + mu \mid \text{input} \mid) / \log(1 + mu)$
`DTMFGenerator`	Generate a dual-tone modulated-frequency (DTMF) signal by adding a low frequency and a high frequency sinusoid together. DTMF tones only consist of first harmonics. The default parameters generate a "1" on a touchtone telephone.
`PostTest`	Return whether or not a valid dual-tone modulated-frequency has been correctly detected based on the last three detection results.
`ToneStrength`	Decision circuit for dual-tone modulated-frequency (DTMF) decoding. It returns true if *Amax* is greater than or equal to A_i for i = 1, 2, 3, 4 such that i does not equal *index*.

Touchtone Decoders

`DTMFDecoder`	Dual-tone modulated-frequency (DTMF) decoder based on post-processing of a bank of Goertzel discrete Fourier transform filters. This galaxy decodes touch tones generated by a telephone.
`DTMFDecoderBank`	Implement one of the banks for detecting dual-tone frequency-

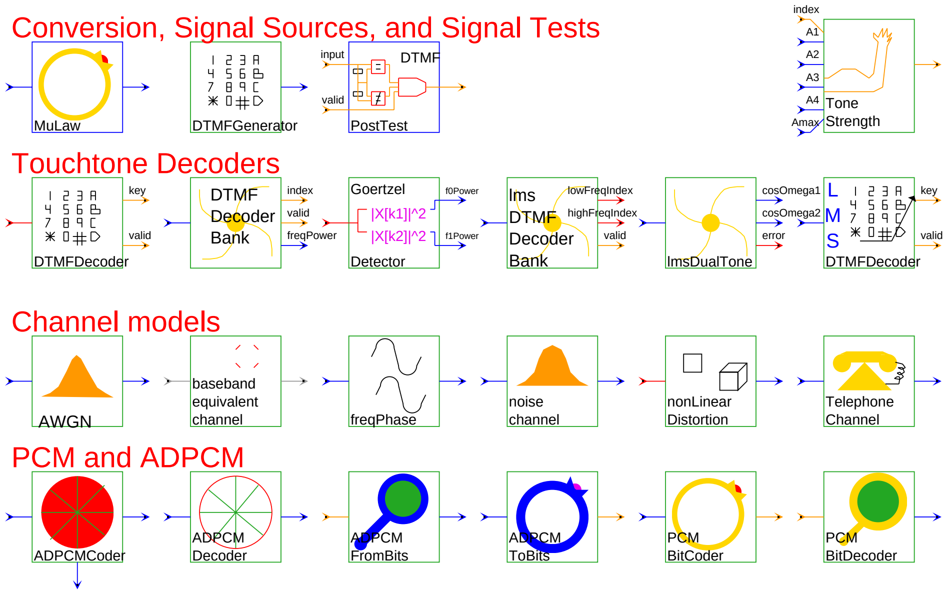


FIGURE 5-16: The palette of telecommunications stars for the SDF domain.

	modulated (DTMF) touch tones. Touch tones are generated by adding a low frequency and a high frequency sinusoid together. The galaxy is used to detect either the low or high frequency component, depending on the parameter settings. This algorithm examines the magnitude of the expected frequency components and their second harmonics. DTMF tones do not have second harmonics, so if they are present, then the input is likely speech and not touch tones. The valid output is true if the input is probably a touch tone. The default parameters are used to detect the low frequency tones.
`GoertzelDetector`	Detect the energy of the first and second harmonic using a pair of Goertzel filters.
`lmsDTMFDecoderBank`	Dual-tone modulated frequency detection based on the post-processing of the output of two LMS algorithms in cascade. These two algorithms are used to detect the two strongest frequencies present in the signal.
`lmsDualTone`	Detect the location of the two strongest harmonic components in the input signal for every input sample using the normalize direct frequency estimation technique, which is based on the LMS algorithm. This galaxy is used in touchtone detection.
`lmsDTMFDecoder`	Least-mean squares dual-tone modulated-frequency decoder. Dual-tone modulated frequency detection based on the post-processing of the output of two LMS algorithms in cascade. These two algorithms are used to detect the two strongest frequencies present in the signal.

Channel Models

For more complete descriptions, see the channel models for the communications stars given on page 5-36.

`AWGN`	Simulate a channel with additive Gaussian noise.
`basebandEquivChannel`	Baseband equivalent channel.
`freqPhase`	Impose frequency offset and/or phase jitter on a signal in order to model channels, such as telephone channels, that suffer these impairments.
`noiseChannel`	A simple channel model with additive Gaussian white noise.
`nonLinearDistortion`	Generate second and third harmonic distortion by squaring and cubing the signal, and adding the results in controlled proportion to the original signal.

`TelephoneChannel`	Telephone channel simulator with Gaussian noise and nonlinear distortion.

PCM and ADPCM

`ADPCMCoder`	Implement adaptive differential pulse code modulation using an `LMS` star. Both the quantized and unquantized prediction-error signals are available as outputs.
`ADPCMDecoder`	Decode the quantized prediction error signal produced by the `ADPCMCoder` galaxy.
`ADPCMFromBits`	Convert a bit stream encoded with the `ADPCMToBits` galaxy back to floating-point values. The 4 low-order bits of the input integer are changed to 1 of 16 floating-point values scaled by *range*.
`ADPCMToBits`	Convert the quantized prediction error of the `ADPCMCoder` galaxy into a bit stream. The quantized prediction error has 16 possible levels, so this galaxy produces 4 bits in each output sample.
`PCMBitCoder`	64kps PCM encoder (CCITT Recommendation G.711).
`PCMBitDecoder`	64kps PCM encoder (CCITT Recommendation G.711).

5.2.15 Spatial Array Processing

The spatial array processing stars given here support a single demonstration named `RadarChainProcessing` developed by Karim Khiar from Thomson CSF. The radar simulation, though five-dimensional, is implemented using SDF, which is a one-dimensional dataflow model. The stars on this palette are shown in figure 5-17.

Data Models

`RadarAntenna`	Generate a specified number of Doppler filter outputs. This galaxy consists of a cascade of a network of antennas, a bank of matched filters, a bank of windows, and a Doppler filter. The bank of matched filters convolves the antenna outputs with a filter matched to a complex pulse train.
`RadarTargets`	Model the observed data as the addition of the receive signal plus sensor noise. The received signal consists of a summation of the emissions of all of the targets.
`GenTarget`	Model the reception of signals by one sensor. A complex pulse train is delayed and then multiplied by a complex exponential.
`RectCx`	Generate a rectangular pulse of width “width” (default 240). If “period” is greater than zero, then the pulse is repeated with the given period.

Sensor and Antenna Models

`SubAntenna`	Models a subantenna. It multiplies the input by a complex exponential.
`sensor`	Compute the excitation of a plane wave arriving at a sensor at the given position with the arrival angle specified as an input. Position (0,0) is assumed to receive phase zero for any angle of arrival.
`ThermalNoise`	Generate thermal noise as a complex noise process whose real and imaginary components are identically independently distributed Gaussian random processes.
`Psi`	Model subantenna excitation.
`SpheToCart`	Compute the inner product of two vectors, one given by a magnitude and two angles in spherical components, the other given by three cartesian components.

Doppler Effects

`PulseComp`	This galaxy generates any number of targets and performs pulse compression. It uses the original chirp to perform the pulse compression. This output represents the output of the radar processing along the range bin axis. The y-axis represents the target magnitude on a linear, logarithmic scale.
`OneDoppler`	Generate one Doppler output. This galaxy performs an antenna

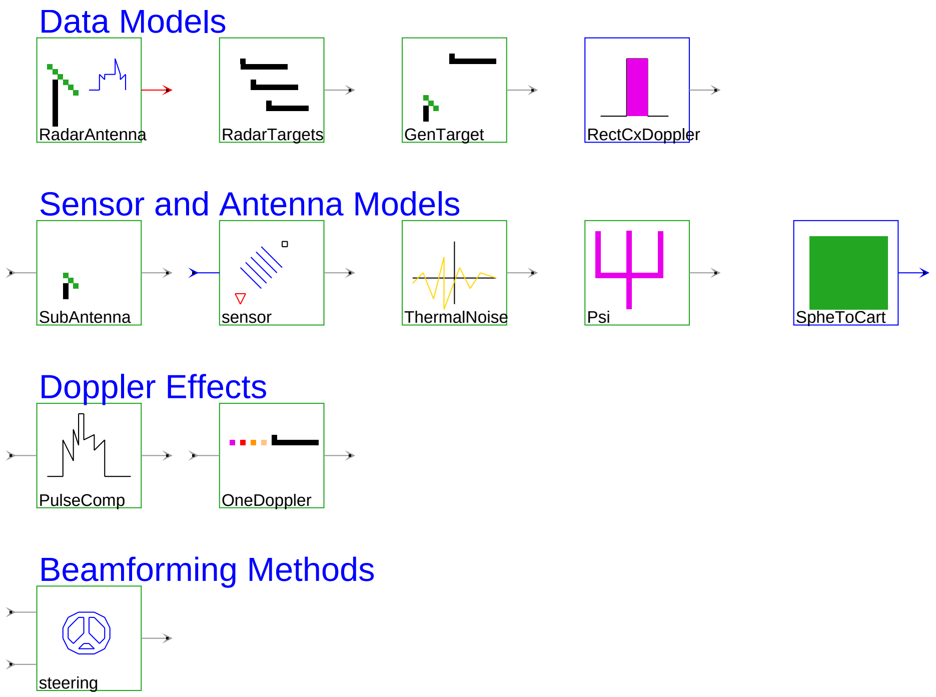


FIGURE 5-17: Spatial Array stars in the SDF domain.

to pulse multiprojection transformation followed by a decimator.

Beamforming Methods

`steering`	Multiply a sensor signal by a window sample and apply a steering correction.

5.2.16 Image processing stars

The image processing stars contained in the palette in figure 5-18 were originally written by Paul Haskell. For the Ptolemy 0.6 release, the image processing infrastructure was rewritten by Bilung Lee to use matrices as the underlying image representation. Since the stars are using the Matrix particle now, some old stars that are just doing simple matrix operation, such as `SumImage`, are removed, and we can use the matrix stars instead, such as `Add_M`.

Displaying images

`DisplayImage`	Accept a black-and-white input grayimage represented by a float matrix and generate output in PGM (portable graymap) format. Send the output to a user-specified command (by default, `xv` is used). The user can set the root filename of the displayed image (which will probably be printed in the image display window

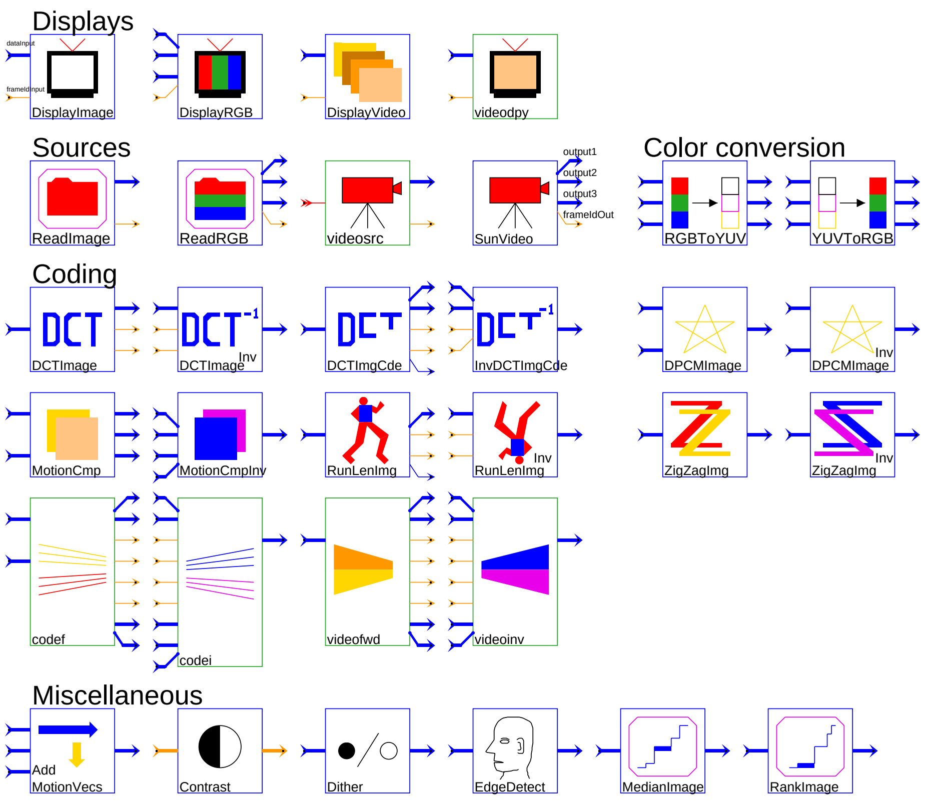


FIGURE 5-18: The Image processing palette in the SDF domain.

	title bar) and can choose whether or not the image file is saved or deleted. The image frame number is appended to the root filename in order to form the complete filename of the displayed image.
`DisplayRGB`	This is similar to `DisplayImage`, but accepts three color images (Red, Green, and Blue) from three input float matrix and generates a PPM (portable pixmap) format color image file. The image file is displayed using a user-specified command (by default, `xv` is used).
`DisplayVideo`	Accept a stream of black-and-white images from input float matrix, save the images to files, and display the resulting files as a moving video sequence. This star requires that programs from the Utah Raster Toolkit (URT) be in your path. Although this toolkit is not included with Ptolemy, it is available for free. See this star's long description (with the "look-inside" or "manual" commands in the Ptolemy menu) for information on how to get the toolkit. The user can set the root filename of the displayed images (which probably will be printed in the display window title bar) with the *ImageName* parameter. If no filename is set, a default will be chosen. The *Save* parameter can be set to `YES` or `NO` to choose whether the created image files should be saved or deleted. Each image's frame number is appended to the root filename in order to form the image's complete filename. The *ByFields* parameter can be set to either `YES` or `NO` to choose whether the input images should be treated as interlaced fields that make up a frame or as entire frames. If the inputs are fields, then the first field should contain frame lines 1, 3, 5, etc. and the second field should contain lines 0, 2, 4, 6, etc.
`videodpy`	Display an image sequence in an X window. This is simply the `SDFDisplayVideo` star encapsulated in a galaxy so that it can be easily used in other domains.

Reading images

`ReadImage`	Read a sequence of PGM-format images from different files and send them out in a float matrix. If present, the character # in the *fileName* parameter is replaced with the frame number to be read next. For example, if the *frameId* parameter is set to 2 and if the *fileName* parameter is `dir.#/pic#` then the file that is read and output is `dir.2/pic2`.

`ReadRGB`	Read a PPM-format image from a file and send it out in three different images— a Red, Green, and Blue image. Each image is represented in a float matrix. The same mechanism for reading successive frames as in `ReadImage` is supported.
`videosrc`	Read in an image from a specified file. This is simply the `SDFReadImage` star encapsulated in a galaxy so that it can be easily used in other domains.
`SunVideo`	Reads frames from the SunVideo card and outputs them as 3 matrices: one for Y,U and V components. This star is new in Ptolemy 0.7, and does not yet have any demos.

Color conversions

`RGBToYUV`	Read three float matrices that describe a color image in RGB format and output three float matrices that describe an image in YUV format. No downsampling is done on the U and V signals.
`YUVToRGB`	Read three float matrices that describe a color image in YUV format and output three float matrices that describe an image in RGB format.

Image and video coding

`DCTImage`	Take a float matrix input particle, compute the discrete cosine transform (DCT), and output a float matrix.
`DCTImageInv`	Take a float matrix input, compute the inverse discrete cosine transform (DCT), and output a float matrix.
`DCTImgCde`	Take a float matrix which represents a DCT image, insert “start of block” markers, run-length encode it, and output the modified image. For the run-length encoding, all values with absolute value less than the *Thresh* parameter are set to 0.0, to help improve compression. Runlengths are coded with a “start of run” symbol and then an (integer) run-length. The *HiPri* parameter determines the number of DCT coefficients per block are sent to “hiport”, the high-priority output. The remainder of the coefficients are sent to “loport”, the low-priority output.
`InvDCTImgCde`	Read two coded float matrices (one high priority and one low-priority), invert the run-length encoding, and output the resulting float matrix. Protection is built in to avoid crashing even if some of the coded input data is affected by loss.
`DPCMImage`	Implement differential pulse code modulation of an image. If the “past” input is not a float matrix or has size 0, pass the “input” directly to the “output”. Otherwise, subtract the “past”

	from the “input” (with leakage factor *alpha*) and send the result to “output”.
`DPCMImageInv`	This star inverts differential pulse code modulation of an image. If the “past” input is not a float matrix or has size 0, pass the “diff” directly to the “output”. Otherwise, add the “past” to the “diff” (with leakage factor *alpha*) and send the result to “output”.
`MotionCmp`	If the “past” input is not a float matrix (e.g. `dummyMessage`), copy the “input” image unchanged to the “diffOut” output and send a null field (zero size matrix) of motion vectors to “mvHorzOut” and “mvVertOut” outputs. This should usually happen only on the first firing of the star. For all other inputs, perform motion compensation and write the difference frames and motion vector frames to the corresponding outputs. This star can be used as a base class to implement slightly different motion compensation algorithms. For example, synchronization techniques can be added or reduced-search motion compensation can be performed.
`MotionCmpInv`	For NULL inputs (zero size matrices) on “mvHorzIn” and/or “mvVertIn”, copy the “diffIn” input unchanged to “output” and discard the “pastIn” input. (A NULL input usually indicates the first frame of a sequence.) For non-NULL “mvHorzIn” and “mvVertIn” inputs, perform inverse motion compensation and write the result to “output”.
`RunLenImg`	Accept a float matrix and run-length encode it. All values closer than *Thresh* to *meanVal* are set to *meanVal* to help improve compression. Run lengths are coded with a start symbol of *meanVal* and then a run-length between 1 and 255. Runs longer than 255 must be coded in separate pieces.
`RunLenImgInv`	Accept a float matrix and inverse run-length encode it.
`ZigZagImage`	Zig-zag scan a float matrix and output the result. This is useful before quantization.
`ZigZagImageInv`	Inverse zig-zag scan a float matrix.
`codef`	This galaxy encodes a sequence of images using motion compensation, a discrete-cosine transform, quantization, and run-length encoding. The outputs are split into high priority and low priority, where corruption of the low priority data will impact the image less.
`codei`	This galaxy inverts the encoding of the codef block, and outputs a reconstructed image sequence.

`videofwd`	This galaxy is obsolete and will probably disappear in the next release.
`videoinv`	This galaxy is obsolete and will probably disappear in the next release.

Miscellaneous image blocks

`AddMotionVecs`	Over each block in the input image, superimpose an arrow indicating the size and direction of the corresponding motion vector.
`Contrast`	Enhance the contrast in the input image by histogram modification. Input image should be in an integer matrix. The possible contrast type are `Uniform` (default) and `Hyperbolic`.
`Dither`	Do digital halftoning (dither) of input image for monochrome printing. Input image should be in a float matrix. The possible dither methods are `Err-Diffusion` (default), `Clustered`, `Dispersed`, and `Own`. If you specify `Own`, then you can use your own dither mask.
`EdgeDetect`	Detect edges in the input image. Input image should be in a float matrix. The possible detectors are `Sobel` (default), `Roberts`, `Prewitt`, and `Frei-Chen`.
`MedianImage`	Accept an input grayimage represented by a float matrix, median-filter the image, and send the result to the output. Filter widths of 1, 3, 5 work well. Any length longer than 5 will take a long time to run. Median filtering is useful for removing impulse-type noise from images. It also smooths out textures, so it is a useful pre-processing step before edge detection. It removes inter-field flicker quite well when displaying single frames from a moving sequence.
`RankImage`	Accept an input grayimage represented by a float matrix, rank filter the image, and send the result to the output. A common example of a rank filter is the median filter, e.g. `MedianImage`, which is derived from this star. Pixels at the image boundaries are copied and not rank filtered.

5.2.17 Neural Networks

The neural network stars demonstrate logic functions using classical artificial neurons and McCulloch-Pitts neuron. These stars were written by Biao Lu (The University of Texas at Austin), Brian L. Evans (The University of Texas at Austin), and are present in Ptolemy 0.7

and later.The neural network stars are shown in figure

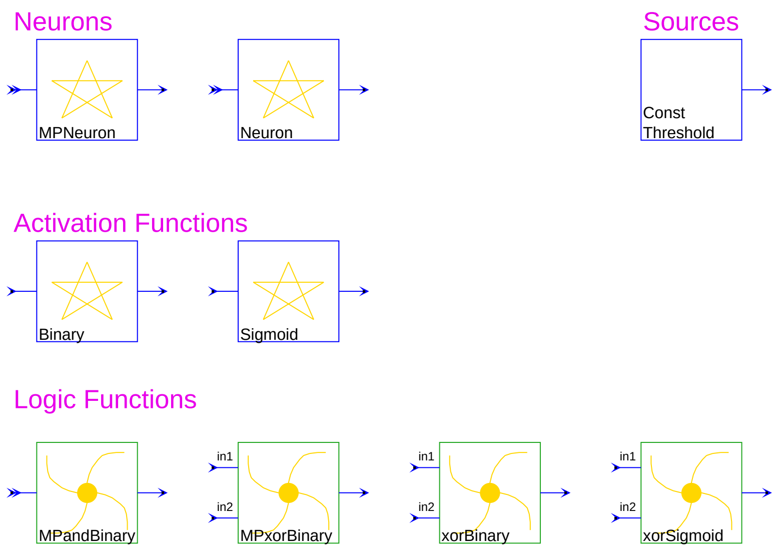


FIGURE 5-19: Neural network stars in the SDF domain.

`MPNeuron`	This is a McCulloch-Pitts neuron. The activation of this neuron is binary. That is, at any time step, the neuron either fires, or does not fire.
`Neuron`	This neuron will output the sum of the weighted inputs, as a floating value.
`ConstThreshold`	Output a constant signal with value given by the “level” parameter (default 0.0)
`Binary`	Binary threshold of the input.
`Sigmoid`	Compute the Sigmoid function, defined as 1/(1 + exp(-r*input)), where r is the learning rate.
`MPandBinary`	The fact that the McCulloch-Pitts neuron is a digital device makes this neuron well-suited to the representation of a two-valued logic, such as AND, OR, and NAND.
`MPxorBinary`	This example shows that a network of McCulloch-Pitts neurons has the power of the finite state automaton known as a Turing machine.
`xorBinary`	XOR function can be implemented by a three-layer neural network which consists of an input layer, a hidden layer and an output layer. A binary activation function is used.
`xorSigmoid`	XOR function can be implemented by a three-layer neural network which consists of an input layer, a hidden layer and an output layer. A sigmoid activation function is used.

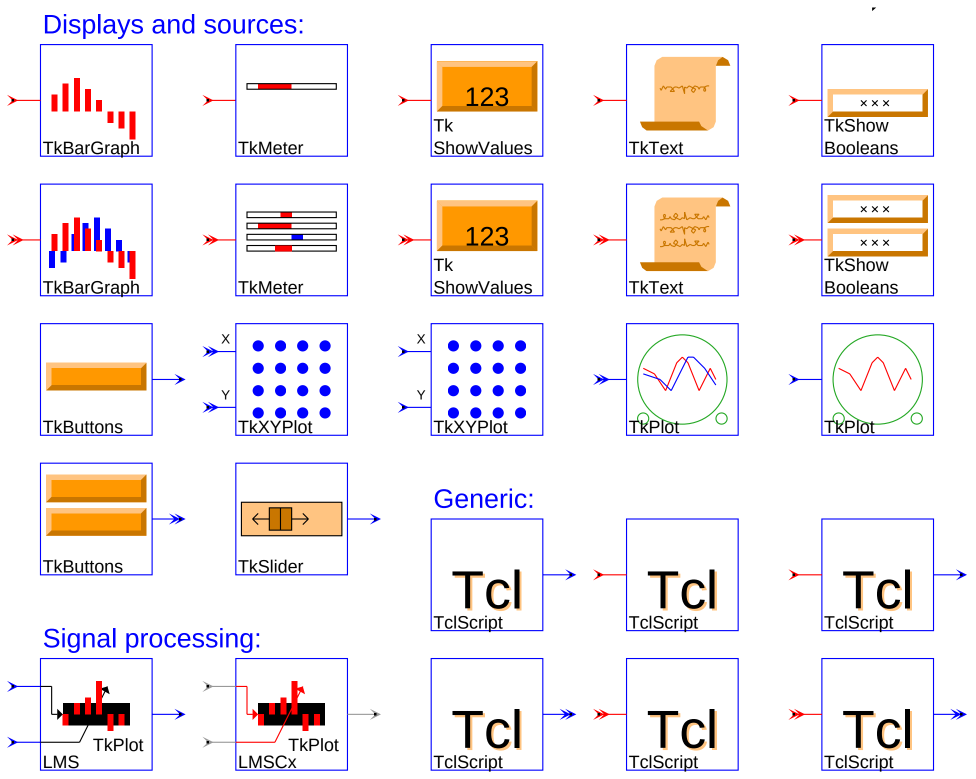


FIGURE 5-20: The Tcl/Tk palette includes many stars that also appear in other palettes.

5.2.18 Higher Order Function stars

The Higher Order Function stars are documented in “An overview of the HOF stars” on page 6-15.

5.2.19 User Contributions

The User Contributed stars are not documented at this time. These stars have been contributed by various users as proofs of concepts. They cannot be retargeted to code generation domains, and we may in the future choose not to release them.

5.2.20 Tcl stars

Most of the stars that interface to Tcl appear in palettes that reflect their function. For instance, all the stars beginning with `Tk` in the “sinks” palette are actually Tcl stars derived from `TclScript`. This is the most generic Tcl star, with no useful function on its own. It must have a Tcl script associated with it to make it useful. There is a chapter of the Programmer’s Manual of the The Almagest devoted to how to write such scripts. The complete palette of Tcl stars, which includes many stars that also appear in other palettes, is shown in figure 5-20. These stars, although derived from `TclScript`, assume the presence of the Tk graphics toolkit. For descriptions of the display and “sink” stars, see Sections 5.2.1 and 5.2.2, respectively.

5.3 An overview of SDF demonstrations

A rather large number of SDF demonstrations have been developed. These can serve as valuable illustrations of the possibilities. Almost every star is illustrated in the demos. Because of the large number, the demos are organized into a set of palettes. Certain demos may appear in more than one palette. A top-level palette, shown in figure 5-21, contains an icon for each demo palette. Notice that the demo palettes collect the hierarchy in a single column, whereas the star palettes collect the hierarchy in two columns.

5.3.1 Basic demos

These demos illustrate the use of certain stars without necessarily performing functions that are sophisticated. The palette is shown in figure 5-22. The demos are described below from left to right, top to bottom.

`butterfly`	Use sines and cosines to compute a curve known as the butterfly curve, invented by T. Fay. The curve is plotted in polar form.
`chaoticNoise`	Chaotic Markov map example with a nonlinear feedback loop.
`comparison`	Compare two sinusoidal signals using the `Test` star.

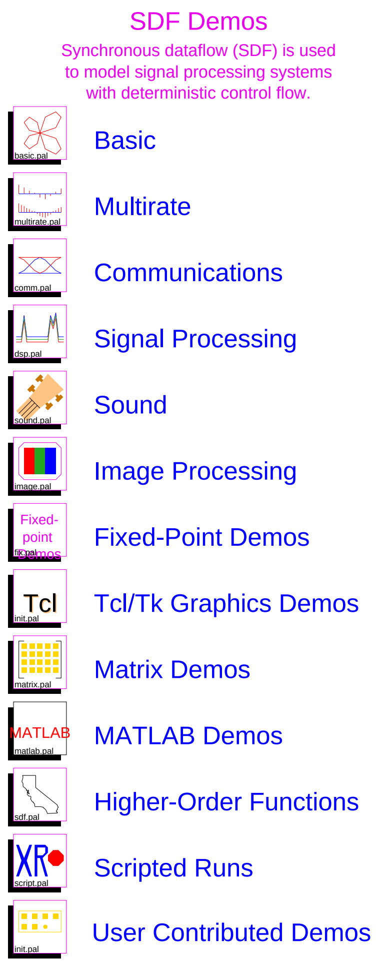


FIGURE 5-21: The top-level demo palette for the SDF domain.

`complexExponential`	Generate and plot a complex exponential.
`delayTest`	Illustrates the use of initializable delays.
`lmsFreqDetect`	Illustrate the use of the LMS algorithm to estimate the dominant sinusoidal frequency in the input signal.
`freqPhaseOffset`	Impose frequency jitter and phase offset on a sinusoid using the `freqPhase` SDF block.
`gaussian`	Generate a Gaussian white noise signal, and plot its histogram and estimated autocorrelation.
`integrator`	Demonstrate the features of the integrator star, such as limiting, leakage, and resetting.
`Modulo`	Demonstrate modulus computation for float and integer data types.
`muxDeMux`	Demonstrate the `Mux` and `DeMux` stars, which perform multiplexing and demultiplexing. Contrast with the `scramble` demo below.
`quantize`	Demonstrate the use of the `Quantizer` star.
`scramble`	This system rearranges the order of samples of signal using the `Commutator` and `Distributor` stars. Note that because these are multirate stars, one iteration involves more than one sample. Contrast with the `muxDeMux` demo above.
`sinMod`	Modulate a sinusoid by multiplying by another sinusoid.

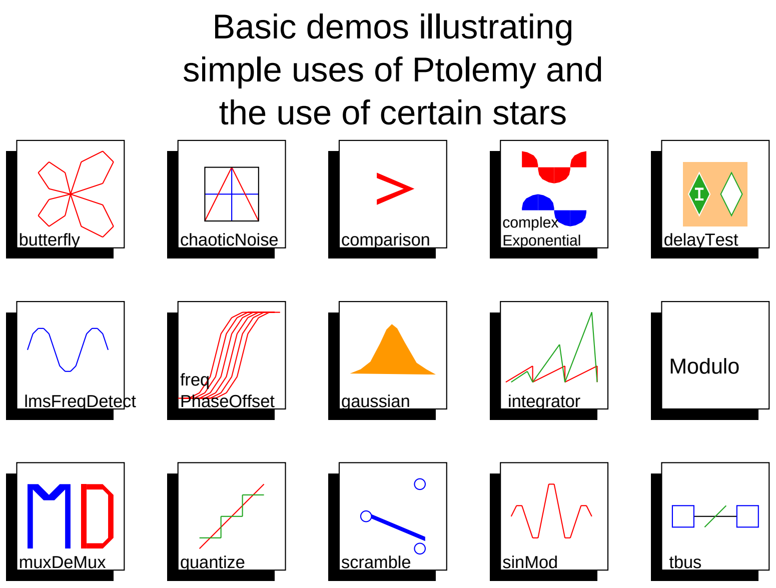


FIGURE 5-22: Palette for a set of basic demos for the SDF domain.

`tbus`	Illustrate the bus facility in Ptolemy, in which multiple signals are combined onto a single graphical connection.

5.3.2 Multirate demos

The demos with icons shown in figure 5-23 illustrate synchronous dataflow principles as applied to multirate signal processing problems. These are arranged roughly in order of sophistication.

`analytic`	Use a `FIRCx` star filter to reduce the sample rate of a sinusoid by a factor of 8/5, and at the same time produce a complex approximately analytic signal (one that has no negative frequency components).
`broken`	Give an example of an inconsistent SDF system. It fails to run, generating an error message instead.
`downSample`	Convert from the digital audio tape sampling rate (48 kHz) to the compact disc sampling rate (44.1 kHz). The conversion is performed in multiple stages for better performance.
`filterBank`	Implement an eight-level perfect reconstruction one-dimensional filter bank based on the biorthogonal wavelet decomposition.
`filterBank-NonUniform`	Implement a simple split of the frequency domain into two non-uniform frequency bands.
`interp`	Use an FIR filter to upsample by a factor of 8 and linearly interpolate between samples.
`multirate`	Upsample a sinusoidal signal by a ratio of 5/2 using a polyphase lowpass interpolating FIR filter.
`upSample`	Convert from the compact disc sampling rate (44.1 kHz) to the digital audio tape sampling rate (48 kHz). The conversion is performed in multiple stages for better performance.

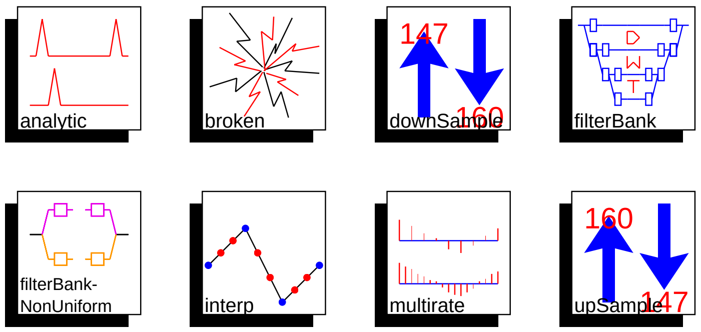


FIGURE 5-23: Multirate signal processing demos in the SDF domain.

5.3.3 Communications demos

The palette shown in figure 5-24 points to some examples of digital communication systems and channel simulators. This palette has been steadily growing.

`constellation`	A 16-QAM signal is sent through a baseband equivalent channel that simulates the following impairments: frequency offset, phase jitter and white Gaussian noise.
`DTMFCodec`	Dual-Tone Modulated Frequency Demo. Generate touch tones and decode the based on the Goertzel Algorithm.
`eye`	Plot an eye diagram for a binary antipodal signal with a raised-cosine pulse shape and user controlled noise.
`lmsDTMFCodec`	Dual-Tone Modulated Frequency Demo. Generate touch tones and decode them based on the LMS Algorithm.
`lossySpeech`	Illustrate the effect on speech of a zero-substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability. Note that this demo requires audio capability and will probably only work on Sun workstations.
`lossySpeechPrevCell`	Illustrate the effect on speech of a previous cell substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability. Note that this demo requires audio capability and will probably only work on Sun workstations.
`modem`	Baseband model of a 16-QAM modem.
`pseudoRandom`	Generate a pseudo-random sequence of zeros and ones using a maximal-length shift register and test its randomness by estimating it autocorrelation.
`pulses`	Generate raised cosine and square-root raised cosine pulses and demonstrate matched filtering with the square-root raised cosine pulse.

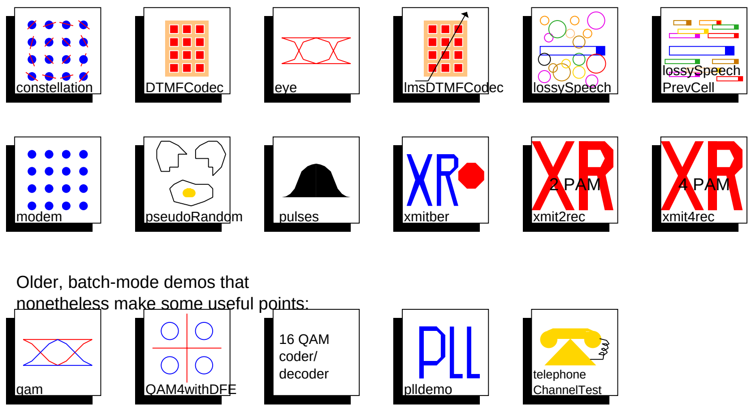


FIGURE 5-24: Communication system demos in the SDF domain.

`xmitber`	Bit Error determination through simulation at various noise levels.
`xmit2rec`	Simple 2-level PAM communication system (matched filtering at the receiver).
`xmit4rec`	Simple 4-level PAM communication system (no filtering at the receiver).

Older communications demos

`qam`	Produce a 16-point quadrature amplitude modulated (QAM) signal and displays the eye diagram for the in-phase part, the constellation, and the modulated transmited signal.
`QAM4withDFE`	This is a model of a digital communication system that uses quadrature amplitude modulation (QAM) and a fractionally spaced decision feedback equalizer.
`codeDecode`	Encode and decode a 16-QAM signal using differential encoding for the quadrant and Gray coding for the point within the quadrant.
`plldemo`	Simulate a fourth-power optical phase-locked loop with laser phase noise and additive Gaussian white noise operating on a complex baseband envelope model of the signal.
`telephoneChannelTest`	Assuming a sampling rate of 8 kHz, a sinusoid at 500 Hz is transmitted through a simulation of a telephone channel with additive Gaussian noise, nonlinear distortion, and phase jitter.

5.3.4 Digital signal processing demos

A fairly large number of signal processing applications are represented in the palette shown in figure 5-25. Several of these serve as good examples to help in solving the exercises included at the end of the chapter.

`adaptFilter`	An LMS adaptive filter converges so that its transfer function matches that of a fixed FIR filter.
`allPole`	Two realizations of an all-pole filter are shown to be equivalent. One uses an FIR filter in a feedback path, the other uses the `BlockAllPole` star.
`animatedLMS`	An LMS adaptive filter is configured as in the `adaptFilter` demo, but this time the filter taps are displayed as they adapt.
`animatedLMSCx`	A complex LMS adaptive filter is configured as in the `adapt-Filter` demo, but in addition, user-controlled noise is added to the feedback loop using an on-screen slider to control the amount of noise. The filter taps are displayed as they adapt.
`cep`	Given the coefficients of any polynomial, this demo uses the

	cepstrum to find a minimum-phase polynomial. Thus, given the coefficients of the denominator polynomial of an unstable filter, this demo will compute the coefficients of a stable denominator polynomial that has the same magnitude frequency response.
`chaos`	This is a simple demonstration of chaos, in which the phase-space plot of the famous Henon map is given.
`convolve`	Convolve two rectangular pulses in order to demonstrate the `Convolve` star.
`dft`	Compute a discrete Fourier transform of a finite signal using the `FFT` star. The magnitude and phase (unwrapped) are plotted.
`doppler`	A sine wave is subjected to four successive amounts of doppler shift. The doppler shift is accomplished by the `phaseShift` galaxy, which forms an analytic signal (using a Hilbert transform) that modulates a complex exponential.
`dtft`	Demonstrate the `DTFT` star, showing how it is different from the `FFTCx` star. Specifically, the range, number, and spacing of frequency samples is arbitrary.
`freqsample`	This system designs FIR filters using the frequency sampling

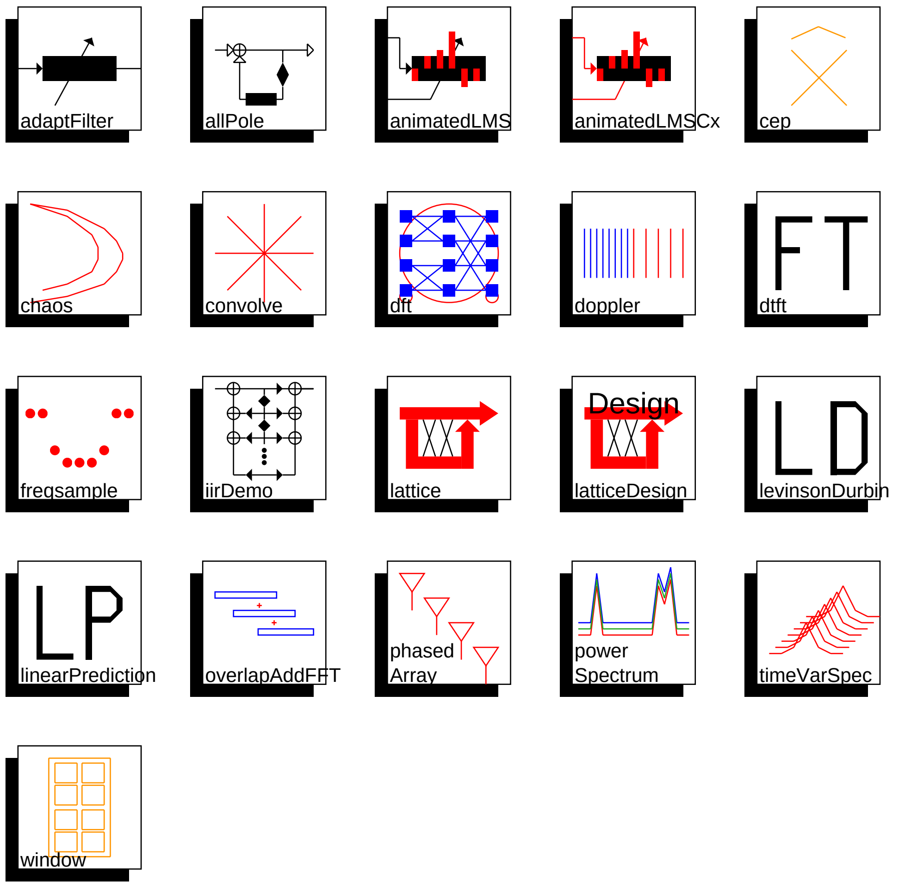


FIGURE 5-25: Signal processing applications in the SDF domain.

	method. Samples of the frequency response are converted into FIR filter coefficients.
`iirDemo`	Two equivalent implementations of IIR filtering.
`lattice`	Demonstrate the use of lattice filters to synthesize an auto-regressive (AR) random process.
`latticeDesign`	Use of Levinson-Durbin algorithm to design a lattice filter with a specified transfer function.
`levinsonDurbin`	Use the Levinson-Durbin algorithm to estimate the parameters of an AR process.
`linearPrediction`	Perform linear prediction on a test signal consisting of three sinusoids in colored, Gaussian noise. Two mechanisms (Burg's algorithm and an LMS adaptive filter) for linear prediction are compared.
`overlapAddFFT`	Convolution is implemented in the frequency domain using overlap and add.
`phasedArray`	Simulate a plane wave approaching a phased array with four sensors. The plane wave approaches from angles starting from head on and slowly rotating 360 degrees. The response of the antenna is plotted as a function of direction of arrival in polar form.
`powerSpectrum`	Compare three methods for estimating a power spectrum of a signal with three sinusoids plus colored noise. The three methods are the periodogram method, the autocorrelation method, and Burg's method.
`timeVarSpec`	A time-varying spectrum is computed using the autocorrelation method and displayed using a waterfall plot.
`window`	Generate and display four window functions and the magnitude of their Fourier transforms. The windows displayed are the Hanning, Hamming, Blackman, and steep Blackman.

5.3.5 Sound-making demos

The demos in the palette in figure 5-26 assume that a program called `ptplay` is in your path, and that it accepts data of an appropriate format and will play it over a workstation speaker at an 8 kHz sample rate. If you are using a Sun SPARCStation, these conditions will most likely be satisfied, if your path is correct. The `ptplay` program has also been used on SGI Indigos and HP 700s and 800s. If you are on an HP, you may need other publicly available software.The samples are written into a file before they are played. Since a large number of samples must be generated, these demos can take some time to run. By contrast, the CGC domain has some audio demos that generate sounds in real time at 44.1kHz, assuming a reasonably fast workstation. For further information about playing audio files, see “Sounds” on page 2-38.

`chirpplay`	Chirp generator that plays on the workstation speaker.
`fmplay`	Sound generator using FM modulation that plays on the workstation speaker.
`speech`	Read a speech signal from a file, and encode it at two bits per sample using adaptive differential pulse code modulation with a feedback-around-quantizer structure. The signal is then reconstructed from the quantized data. The original and reconstructed speech are played over the workstation speaker.
`KSchord`	Simulation of plucked string sounds using the Karplus-Strong algorithm.
`vox`	Coarticulation with an Adaptive Vocoder. The resulting FM synthesized sound is played over the workstation speaker.
`blockVox`	A block processed version of the vox demo.
`lossySpeech`	Illustrate the effect on speech of a zero-substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability. This demo also appears in the basic demos palette
`lossySpeechPrevCell`	Illustrate the effect on speech of a previous cell substitution pol-

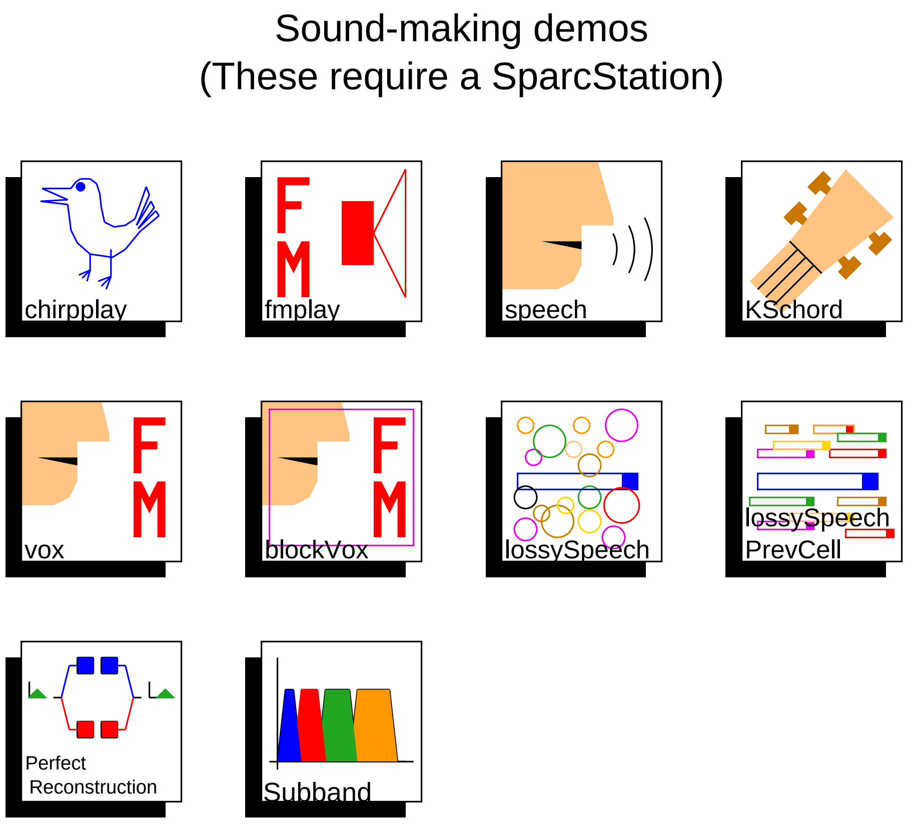


FIGURE 5-26: Sound-making demos in the SDF domain.

icy in a network (such as ATM) with 48 byte packets and a variable loss probability. This demo also appears in the basic demos palette.

`perfectReconstuction`	Eight-channel perfect reconstruction one-dimensional analysis/synthesis filterbank. The incoming speech signal is split into eight adjacent frequency bins and then reconstructed. The original and reconstructed speech are played over the workstation speaker.
`subbandcoding`	Four channel subband speech coding with APCM at 16kps.

5.3.6 Image and video processing demos

The demos in figure 5-27 all read images from files on the workstation disk, process them, and then display them. Some of the demos process short sequences of images, thus illustrating video processing in Ptolemy. They all use the image classes described in “Image processing stars” on page 5-44. The set of demos in this palette does not reflect the richness of possibilities. See the DE domain for more image and video signal processing applications in the context of packet-switched network simulations. The video display requires that the Utah Raster Toolkit be installed and available in the user’s path.

`BlendImage`	Combine two images and display the result.
`bwDither`	Demonstrate four different forms of black and white dithering: error diffusion, clustered dither, dispersed dither, and use custom mask.

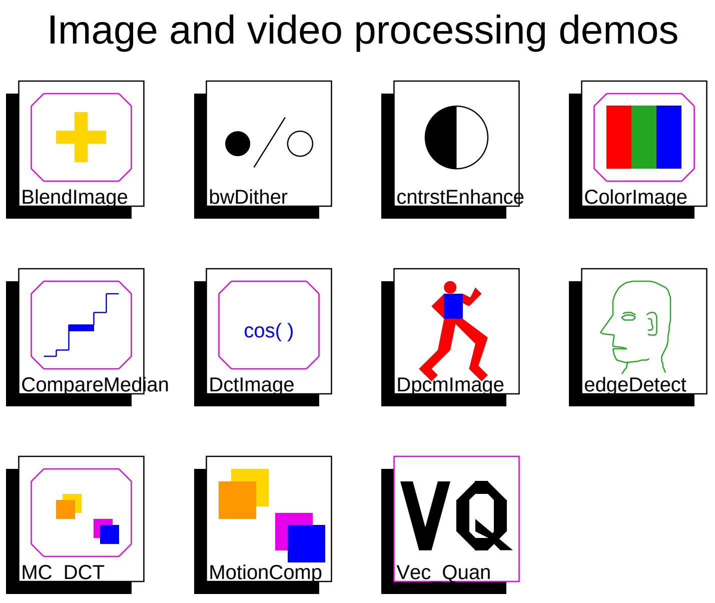


FIGURE 5-27: Image processing demos in the SDF domain.

`cntrastEnhance`	Contrast enhancement by histogram modification.
`ColorImage`	Convert an RGB (red-green-blue) format color image to YUV (luminance-hue-saturation) format and back, and then display it on the workstation screen.
`CompareMedian`	Median filter an image to reduce artifacts due to interleaved scanning in video sequences.
`DctImage`	Perform discrete cosine transform (DCT) coding of an image sequence.
`DpcmImage`	Perform differential pulse code modulation (DPCM) on an image sequence.
`EdgeDetect`	Demonstrate four different forms of edge detection: Sobel, Roberts, Prewitt, and Frei-Chen.
`MC_DCT`	Perform motion compensation and DCT encoding of video.
`MotionComp`	Perform motion compensation video coding.

Vector Quantization demonstrations

The `Vec_Quan` icon in the image processing palette brings up a sub-palette that has several vector quantization demonstrations in figure 5-28:

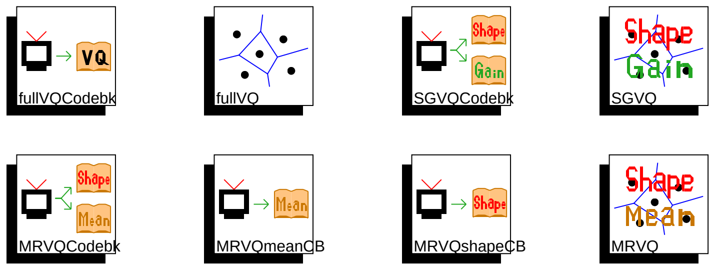


FIGURE 5-28: Vector Quantization demos in the SDF domain

`fullVQCodebk`	Generate a codebook for full search vector quantization.
`fullVQ`	Full search vector quantization using codebook generated by `fullVQCodebk`.
`SGVQCodebk`	Generate codebooks for shape-gain vector quantization.
`SGVQ`	Shape-gain vector quantization using codebook from `SGVQ-Codebk`.
`MRVQCodeBk`	Generate codebooks for mean-removed vector quantization using independent quantizer structure.
`MRVQmeanCB`	Generate codebook for mean-removed vector quantization.
`MRVQshapeCB`	Generate the shape codebook for mean-removed quantization

	using alternate structure. This universe uses the codebook generated by `MRVQmeanCB`.
`MRVQ`	Mean-removed vector quantization.

5.3.7 Fixed-point demos

The demos shown in figure 5-29 illustrate the use of fixed-point stars in the SDF domain.

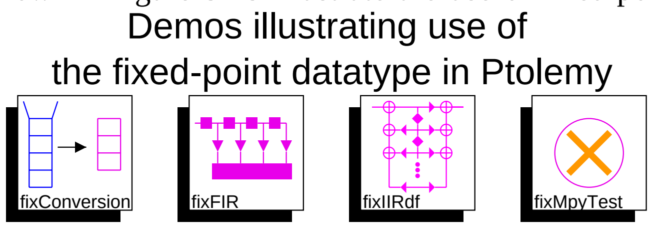

Demos illustrating use of
the fixed-point datatype in Ptolemy
fixConversion
fixFIR
fixIIRdf
fixMpyTest

FIGURE 5-29: These demos illustrate fixed-point effects in signal processing systems.

These stars are used to model hardware implementations with finite precision.

`fixConversion`	Illustration of the different masking options available.
`fixFIR`	Effect of filter tap precision on the frequency response.
`fixIIRdf`	Comparison of a fourth-order direct-form IIR filter implemented with floating-point arithmetic and a similar filter implemented with fixed-point arithmetic.
`fixMpyTest`	Testing of fixed-point multiplication over a range of numbers by comparison against floating-point multiplication. The results should be the same.

5.3.8 Tcl/Tk demos

These demos shown in figure allow the user to interact with the simulation. The interactivity is provided by the Tcl scripting language controlling the Tk graphics toolkit. Tcl is integrated throughout Ptolemy. Tk has been integrated into the graphical user interfaces for Ptolemy, but not in the `ptcl` textual interpreter. Therefore, these stars do not work in `ptcl`.

`animatedLMS`	See “Digital signal processing demos” on page 5-55.
`animatedLMSCx`	See “Digital signal processing demos” on page 5-55.
`buttons`	Demonstrate `TkButtons`.
`phased_Array`	Demonstrate `TkSlider` by creating a vertical array of radar sensors that can be move in the horizontal plane. Note that small movements of the sensors radically change the polar gain plot. This simulation demonstrates the importance of sensor calibration to performance of the sensor array.
`sinWaves`	Demonstrate `TkBarGraph` by generating and displaying a complex exponential.
`tclScript`	Demonstrate `TclScript` by generating two interactive X win-

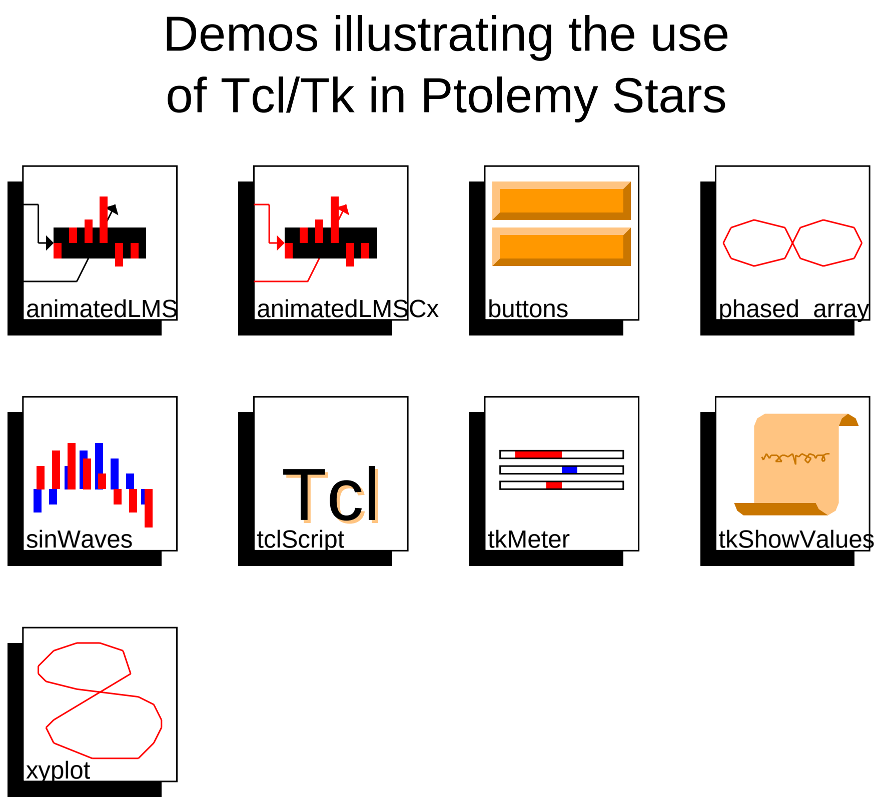


FIGURE 5-30: Tcl/Tk demos in the SDF domain

	dow follies that consist of circles that move in the same playing field.
`tkMeter`	Demonstrate `TkMeter` by creating three bar meters. The first oscillates sinusoidally. The second displays a random number between zero and one. The third displays a random walk.
`tkShowValues`	Demonstrate `TkShowValues` and `TkText` by displaying the ASCII form of two ramp sequences.
`xyplot`	Demonstrate the dynamic plotting capabilities of the `xyplot` star.

5.3.9 Matrix demos

The systems in figure 5-31 demonstrate the use of matrix particles in Ptolemy. Matrices are also used in the SDF domain to represent images. See “Image and video processing demos” on page 5-59. The demonstrations below are primarily to test matrix operations.

`MatrixTest1`	Demonstrate the use of the Matrix stars that have one input. These include the operations inverse, transpose, and multiply by a scalar gain for all matrix types. Also conjugate and Hermitian transpose are available for the complex matrix type.

`MatrixTest2`	Demonstrate the use of some simple Matrix stars with two inputs. These include multiply, add, and subtract.
`MatrixTest3`	Demonstrate the use of the Matrix conversion stars. These convert between the scalar particles and the matrix particles as well as between the various matrix types.
`initDelays`	Illustrate the use of initializable delays with the `matrix` class.
`Kalman_M`	Compare the convergence properties of a Kalman filter to those of an LMS filter when addressing the problem of adaptive equalization of a process in noise.
`SVD_MUSIC_1`	Show the use of singular-value decomposition (SVD) and the Multiple-Signal Characterization (MUSIC) algorithm to identify the frequency of a single sinusoid in a signal that has two different signal to noise ratios.
`SVD_MUSIC_2`	Demonstrate the use of the Multiple-Signal Characterization (MUSIC) algorithm to identify three sinusoids in noise that have frequencies very close to each other.

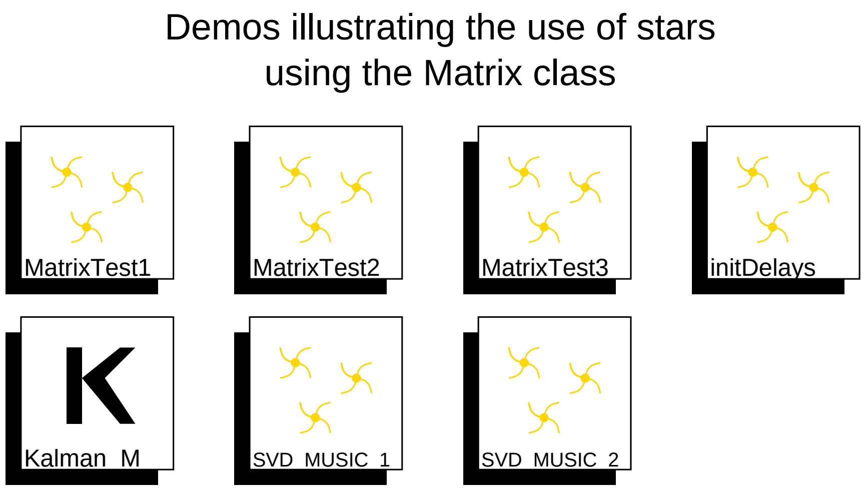


FIGURE 5-31: Demonstrations of matrix operations in Ptolemy.

5.3.10 MATLAB Demos

The demos pictured in figure 5-32 illustrate the use of the MATLAB stars. The MAT-

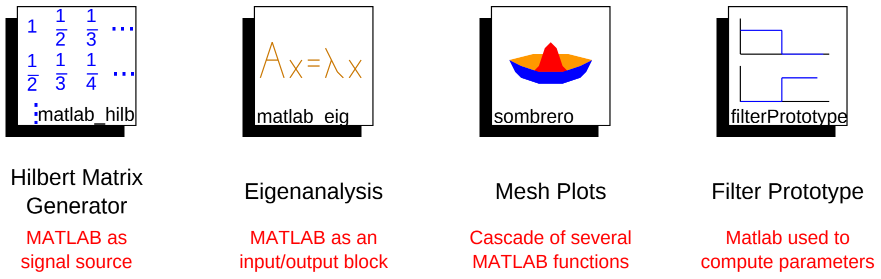


FIGURE 5-32: MATLAB demos in the SDF domain.

LAB stars convert input values into MATLAB matrices, apply a sequence of MATLAB commands to the matrices, and output the result as Ptolemy matrices. The filterPrototype demonstration shows how to use MATLAB to compute parameters of stars. For information about running Matlab on a remote machine, see “Matlab stars” on page 5-26.

`matlab_hilb`	This demo uses MATLAB as a signal source to produce a Hilbert matrix. The Hilbert matrix is an ill-conditioned matrix used to test the robustness of numerical linear algebra routines. The matrix element (i,j) has the value of 1 / (i + j - 1). The matrix values appear similar to the coefficients of a discrete Hilbert transformer.
`matlab_eig`	This demo shows the use of MATLAB to perform eigendecomposition of a 2 x 2 Hermitian symmetric complex matrix. A matrix of eigenvectors and a matrix of eigenvalues are produced. The eigenvalues are real because the input matrix is Hermitian symmetric
`sombrero`	This demo is an entire universe composed of a cascade of four MATLAB stars. The MATLAB stars are used a signal source and a signal sink. The overall system generates and plots a mathematical model of a two-dimensional sinc function that resembles a sombrero.
`filterPrototype`	This system uses a halfband lowpass filter prototype for the lowpass and highpass filters. All parameters are computed using MATLAB.

5.3.11 HOF Demos

The Higher Order Function demos are described in the HOF domain chapter. See “An overview of HOF demos” on page 6-18.

5.3.12 Scripted Runs

A scripted run executes the tcl code in the run control panel tcl script window. Scripted runs can be used to set up interactive tutorials.The demos shown in figure 5-33 illustrate the use of scripted runs.

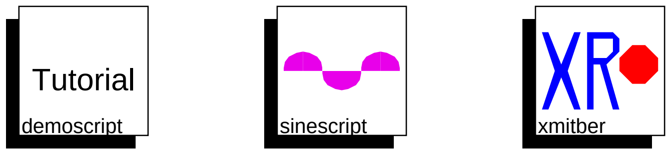


FIGURE 5-33: Scripted run demos in the SDF domain.

`demoscript`	An interactive tutorial that leads a user through a session that runs a simple universe.
`sinescript`	This demo runs the same sine wave modulation universe three times, each time with a different frequency.
`xmitber`	This demo runs a bit error determination universe at various noise levels and then plots the output.

5.4 Targets

As is typical of simulation domains, the SDF domain does not have many targets. To choose one of these targets, with your mouse cursor in a schematic window, execute the `edit-target` command under the `Edit` vem menu choice (or just type “`T`”). You will get a list of the available `Targets` in the SDF domain. The “default-SDF” target is normally selected by default. When you click `OK`, dialog box appears with the parameters of the target. You can edit these, or accept the defaults. The next time you run the schematic, the selected target will be used. For more information, see “Summary of Uniprocessor schedulers” on page 4-11.

5.4.1 Default SDF target

The default SDF target has a simple set of options:

logFile	(`STRING`) Default = The name of a file into which the scheduler will write the final schedule. The initial default is the empty string.
loopScheduler	(`STRING`) Default = DEF A String specifying whether to attempt to compact the schedule for forming looping structure (see below). Choices are DEF, CLUST, ACYLOOP. The case does not matter: DEF, def, Def are all the same. For backward compatibility, “0” or “NO”, and “1” or “YES” are also recognized, with “0” or “NO” being DEF, and “1” or “YES” being CLUST.
schedulePeriod	(`FLOAT`) Default = 0.0 A floating-point number defining the time taken by one iteration through the schedule. This is not needed for pure SDF systems,

but if SDF systems are mixed with timed domains, such as DE, then this will determine the amount of simulated time taken by one iteration.

The SDF scheduler determines the order of execution of stars in a system at start time. It performs most of its computation during its `setup()` phase. If the *loopScheduler* target parameter is DEF, then we get a scheduler that exactly implements the method described in [Lee87a] for sequential schedules. If there are sample rate changes in a program graph, some parts of the graph are executed multiple times. This scheduler does not attempt to generate loops; it simply generates a linear list of blocks to be executed. For example, if star A is executed 100 times, the generated schedule includes 100 instances of A. A loop scheduler will include in its “looped” schedule (where possible) only one instance of A and indicate the repetition count of A, as in (100 A). For simulation, a long unstructured list might be tolerable, but not in code generation. (The SDF schedulers are also used in the code generation for a single processor target).

Neglecting the overhead due to each loop, an optimally compact looped schedule is one that contains only one instance of each actor, and we refer to such schedules as *single appearance schedules*. For example, the looped schedule (3 A)(2 B), corresponding to the firing sequence AAABB, is a single appearance schedule, whereas the schedule AB(2 A)B is not.

By setting the *loopScheduler* target parameter to CLUST, we select a scheduler developed by Joe Buck. Before applying the non-looping scheduling algorithm, this algorithm collects actors into a hierarchy of clusters. This clustering algorithm consists of alternating a “merging” step and a “looping” step until no further changes can be made. In the merging step, blocks connected together are merged into a cluster if there is no sample rate change between them and the merge will not introduce deadlock. In the looping step, a cluster is looped until it is possible to merge it with the neighbor blocks or clusters. Since this looping algorithm is conservative, some complicated looping possibilities are not always discovered. Hence, even if a graph has a single appearance schedule, this heuristic may not find it.

Setting the *loopScheduler* target parameter to ACYLOOP results in another loop scheduler being selected, this one developed by Praveen Murthy and Shuvra ‘Bhattacharyya [Mur96][Bha96]. This scheduler only tackles acyclic SDF graphs, and if it finds that the universe is not acyclic, it automatically resets the *loopScheduler* target parameter to CLUST. This scheduler is optimized for program as well as buffer memory. Basically, for a given SDF graph, there could be many different single appearance schedules. These are all optimally compact in terms of schedule length (or program memory in inline code generation). However, they will, in general, require differing amounts of buffering memory; the difference in the buffer memory requirement of an arbitrary single appearance schedule versus a single appearance schedule optimized for buffer memory usage can be dramatic. Again, in simulation this does not make that much difference (unless really large SDF graphs with large rate changes are being simulated of-course), but in code generation it is very helpful. Note that acyclic SDF graphs always have single appearance schedules; hence, this scheduler will always give single appearance schedules. If the *logFile* target parameter is set, then a summary of internal scheduling steps will be written to that file. Essentially, two different heuristics are used by the ACYLOOP scheduler, called APGAN and RPMC, and the better one of the two is selected. The generated file will contain the schedule generated by each algorithm, the resulting buffer memory requirement, and a lower bound on the buffer memory requirement (called BMLB) over all possible single appearance schedules.

Note that the ACYLOOP scheduler modifies the universe during its computations; hence, scripted runs that depend on the universe remaining in the original state, cannot be used with this scheduler. Since the universe reverts to its original state after a run sequence, the ACYLOOP scheduler will work fine in normal usage.

5.4.2 The loop-SDF target

An exact looping algorithm, available in an alternative target called the `loop-SDF` target, was developed by adding postprocessing steps to the CLUST loop scheduling algorithm. For lack of a better name, we call this technique "SJS scheduling", for the first initials of the designers (Shuvra Bhattacharyya, Joe Buck, and Soonhoi Ha). In the postprocessing, we attempt to decompose the graph into a hierarchy of acyclic graphs [Bha93b], for which a compact looped schedule can easily be constructed. Cyclic subgraphs that cannot be decomposed by this method, called *tightly interdependent subgraphs*, are expanded to acyclic precedence graphs in which looping structures are extracted by the techniques developed in [Bha94a] and extensions to these techniques developed by Soonhoi Ha. This scheduling option is selected when the *loopTarget* is chosen instead of the default SDF target. The target options are:

logFile

schedulePeriod

They have the same interpretation as for the default target, but in the `loop-SDF` target, *schedulePeriod* has an initial default of 10000.0.

When there are sample rate changes in the program graph, the default SDF scheduler may be much slower than the loop schedulers, and in code generation, the resulting schedules may lead to unacceptably large code size. Buck's scheduler provides a fast way to get compact looped schedules for many program graphs, although there are no guarantees of optimality. The somewhat slower SJS scheduler is guaranteed to find a single appearance schedule whenever one exists [Bha93c]. Furthermore, a schedule generated by the SJS scheduler contains only one instance of each actor that is not contained in a tightly interdependent subgraph. However, neither the SJS scheduler nor Buck's scheduler will attempt to optimize for buffer memory usage; this need is met by the ACYLOOP scheduler chosen through the default-SDF target as described above, for acyclic graphs. Algorithms for generating single appearance schedules optimized for buffer memory systematically for graphs that may contain cycles have not yet been implemented.

The looped result can be seen by setting the *logFile* target parameter. That file will contain all the intermediate procedures of looping and the final scheduling result. The loop scheduling algorithms are usually used in code generation domains, not in the simulation SDF domain. Refer to the Code Generation domain documentation for a detailed discussion to the section on "Schedulers" on page 13-6.

5.4.3 Compile-SDF target

A third target in the SDF domain, called `compile-SDF`. Instead of executing a simulation by invoking the `go()` methods of stars from within the Ptolemy process, it generates a C++ program that implements the universe, links it with appropriate parts of the Ptolemy kernel, and then invokes that system. The schedule is constructed statically, so the generated program has no scheduler linked in. Instead, the generated code directly invokes the `go()` methods of the stars. The target parameters are:

directory	(`STRING`) Default= `$HOME/PTOLEMY_SYSTEMS` The directory into which to place the generated code.
LoopingLevel	(`STRING`) Default = ACYLOOP The choices are DEF, CLUST, SJS, or ACYLOOP. Case does not matter; ACYLOOP is the same as AcyLoOP. If the value is DEF, no attempt will be made to construct a looped schedule. This can result in very large programs for multirate systems, since inline code generation is used, where a codeblock is inserted for each appearance of an actor in the schedule. Setting the level to CLUST invokes a quick and simple loop scheduler that may not always give single appearance schedules. Setting it to SJS invokes the more sophisticated SJS loop scheduler, which can take more time to execute, but is guaranteed to find single appearance schedules whenever they exist. Setting it to ACYLOOP invokes a scheduler that generates single appearance schedules optimized for buffer memory usage, as long as the graph is acyclic. If the graph is not acyclic, and ACYLOOP has been chosen, then the target automatically reverts to the SJS scheduler. For backward compatibility, “0” or “NO”, “1”, and “2” or “YES” are also recognized, with “0” or “NO” being DEF, “1” being CLUST, and “2” or “YES” being SJS.
writeSchedule?	(`INT`) Default = NO If the value is YES, then the schedule is written out to a file named`.sched` in the directory named by the *directory* target parameter.

If you wish to try this SDF target, open any of the basic SDF demos in figure 5-22, edit the target to change it to the `compile-SDF` target (in vem, hit `T`), and run the system. You can then examine the source code and makefile that are placed in the specified directory. An executable with the same name as the name of the demo will also be placed in that directory. This is a standalone executable that does not require any part of the Ptolemy system to run (except for the Ptolemy Tcl/Tk startup scripts in `$PTOLEMY/lib/tcl`). For example, if you choose the `butterfly` demo in figure 5-22, your destination directory will contain the following files:

```
butterfly
butterfly.cc
code.cc
make.template
makefile
```

The first of these is executable. Try executing it. You can modify the number of sample points generated using a command-line argument. For example, to generate 1,000 points instead of 10,000, type

```
butterfly 1000
```

The `compile-SDF` target is an example of a code-generation `Target` within the SDF domain. So, in a very fundamental way, a `Target` defines the way a system is executed. The default target is essentially an interpreter. The `compile-SDF` target synthesizes a standalone program and then executes it on the native workstation. It should be viewed merely as an example of the kinds of extensions users can build. More elaborate targets parallelize the code and execute the resulting programs on remote hardware. Targets can be defined by users and can make use of existing Ptolemy schedulers. Knowledgeable users can also define their own schedulers.

The `compile-SDF` target first creates the C++ source code for the current universe in a file of the same name of the universe followed by `.cc`. Then, it copies the C++ code into `code.cc` and builds the `makefile` to compile `code.cc` using the make template file `CompileMake.template` in the `$PTOLEMY/lib` directory. Next, the `compile-SDF` target runs the `makefile` to compile `code.cc` into an executable called `code`. The `compile-SDF` target then renames `code` to the name of the Ptolemy universe. If there is an error reported by `make`, then it is likely that one of make configuration variables is incorrect. The `makefile` includes the configuration makefile for the workstation you are using. The configuration makefiles are in the `$PTOLEMY/mk` directory.

The compile-SDF target has a number of known problems:

- The resulting C++ program is unnecessarily large (a minimum of about half a megabyte) because many unnecessary Ptolemy objects get linked in. You can create a much leaner program using the CGC domain.
- Error messages during the compile or run are sent to the standard output rather than displayed in a window on the screen.
- If you specify a relative directory for the destination directory, instead of the absolute directory as done in the default, then the location of the directory will be relative to the current working directory of the Ptolemy system. It is easy to lose track of what that is.
- Generating code can take quite a bit of time, particularly if a multirate system is used with the `LoopingLevel` parameter set to “0”. Unfortunately, there is no convenient way to interrupt the code generation process.
- Implicit forks are not currently correctly handled. Consequently, whereas the target works for simple systems, more elaborate systems inevitably cause problems.
- The `make` program on your path must be the GNU `make` program.
- If you have changed Ptolemy versions, then it is likely that the make template has also changed. However, Ptolemy will not copy `CompileMake.template` over an existing `make.template`. If you get errors after you have switched versions of Ptolemy, then delete the `make.template` and `makefile` in the destination directory of the `compile-SDF` target.

We would welcome any assistance in fixing these problems.

5.4.4 SDF to PTCL target

The `SDF-to-PTCL` target was introduced in Ptolemy 0.6. This target is substantially incomplete, we give a rough outline below. We hope to complete work on the `SDF-to-PTCL` target in a later release. The `SDF-to-PTCL` target uses `CGMultiInOut` stars to generate abstract ptcl graphs which capture the SDF semantics of a simulation SDF universe. These abstract graphs can then be used to test SDF schedulers.

The ptcl output filename will use the universe name as a prefix, and append `.pt` to the name (e.g., the ptcl output for the `butterfly` demo would be in `butterfly.pt`). Currently the directory that will contain the ptcl output is hardwired to `~/PTOLEMY_SYSTEMS/ptcl/`. You may need to create this directory by hand.

The most interesting aspect about the target is that it collects statistics on the execution time of each star. This is valuable for seeing the relative runtimes of the various stars which can be used in code generation. It collects statistics by running the scheduled universe, accumulating elapsed CPU time totals for each star. This new target does not call the `wrapup` methods of the stars, so you will not see `XGraph` outputs.

5.5 Exercises

The exercises in this section were developed by Alan Kamas, Edward Lee, and Kennard White for use in the undergraduate and graduate digital signal processing classes at U. C. Berkeley. If you are assigned these exercises for a class, you should turn in printouts of well-labeled schematics, showing all non-default parameter values, and printouts of relevant plots. Combining multiple plots into one can make comparisons more meaningful, and can save paper. Use the `XMgraph` star with multiple inputs.

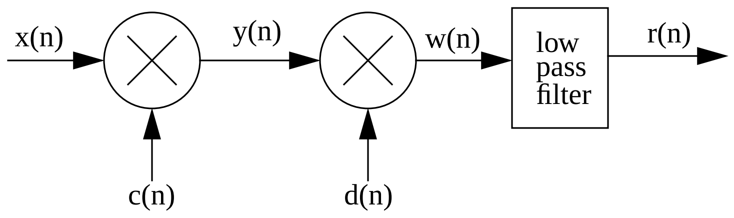
5.5.1 Modulation

This problem explores amplitude modulation (AM) of discrete-time signals. It makes extensive use of FFTs. These will be used to approximate the discrete-time Fourier transform (DTFT). In subsequent exercises, we will study artifacts that can arise from this approximation. For our purposes here, the output of the `FFTCx` block will be interpreted as samples of the DTFT in the interval from 0 (d.c.) to 2π.

Frequencies in many texts are normalized. To make this exercise more physically meaningful, you should assume a sampling frequency of 128 kHz ($T = 7.8\mu\text{sec}$ sampling period). Thus the 0 to 2π range of frequencies (in radians per sample) translates to a range of 0 to 128kHz. On your output graphs, you should clearly label the units of the x-axis. The *xUnits* parameter of the `XMGraph` star can be used to do this. If the FFT produces $N = 256$ samples, representing the range from 0 to $f_s = 128$ kHz., then *xUnits* should be $f_s/N = 500$ Hz. Thus each sample out from the FFT will represent 500Hz. Keep in mind that a DTFT is actually periodic, and that only one cycle will be shown.

With default parameters, the `FFTCx` star will read 256 input samples and produce 256 complex output samples. This gives adequate resolution, so just use the defaults for this exercise. The section “Iterations in SDF” on page 5-3 will tell you, for instance, that you should run your systems for one iteration only. The section “Particle types” on page 2-20 explains how to properly manage complex signals. For this exercise, you should only plot the magnitude of the `FFTCx` output, ignoring the phase.

The overall goal is to build a modulation system that transmits a speech or music signal $x(n)$ using AM modulation. The transmitted signal is $y(n)$. The receiver demodulates y(n) to get the recovered signal $r(n)$. The system is working if $r(n) = x(n)$. Commercial AM radio uses carrier frequencies from 500kHz to 2MHz; however, we will use carriers around 32kHz. This makes the results of the modulation easier to see. The system you will develop (after several intermediate phases) is shown below:



1. The first task is to figure out how to use the `FFTCx` star to plot the magnitude of a DTFT. Begin by generating a signal where you know the DTFT. Use the `Rect` star to generate a rectangular pulse

$$x(n) = \sum_{k=0}^{M} \delta(n-k)$$

   for $M = 4$ and $M = 10$. Plot the magnitude of the DTFT. It would be a good idea at this point to make a galaxy that will output the magnitude of the DTFT of the input signal. Be sure the axis of your graph is labeled with the frequencies in Hz, assuming a sampling frequency of 128kHz.

2. The signal generated above does not have narrow bandwidth. The next task will be to generate a signal $x(n)$ with narrower bandwidth so that the effects of modulating it can be seen more clearly and so there are fewer artifacts. A distinctive and convenient lowpass signal can be generated by feeding an impulse into the `RaisedCosine` star (found in the "communications" palette). Set the parameters of the `RaisedCosine` star as follows:

   length: 256
   symbol_interval: 8
   excessBW: 0.5

   Leave the *interpolation* parameter on its default value. The detailed functionality of this star is not important: we are just using it to get a signal we can work with conveniently. Plot the time domain signal and its magnitude DTFT. What is the bandwidth (single-sided), in Hz, of the signal? Use the -6dB point (amplitude at 1/2 of the peak) as the band edge. The signal was chosen to have roughly the bandwidth of a typical AM broadcast signal.

3. The next task is to modulate the signal $x(n)$ generated in part (2) with a sine wave. Construct a 32 kHz sine wave using the `singen` galaxy and let it be the carrier $c(n)$; then produce $y(n) = x(n)c(n)$. Graph the DTFT of $y(n)$. What is the bandwidth of $y(n)$? Change the carrier to 5 kHz, and graph the FFT of y(n). Explain in words what has happened. Keep the carrier at 5 kHz, and determine what the largest possible bandwidth is for $x(n)$ so that $y(n)$ will not have any significant distortion.

4. The next step it to build the demodulator. First multiply again by the same carrier, $d(n) = c(n)$, and plot the magnitude DTFT of the result. Explain in words what about

this spectrum is directly attributable to the discrete-time nature of the problem. In other words, what would be different if this problem were solved in continuous time?

5. To complete the demodulation, you need to filter out the double frequency terms. Use the `FIR` filter star with its default coefficients. This is not a very good lowpass filter, but it is a lowpass filter. Explain in words exactly how the resulting signal is different from the original baseband signal. How would you make it more like the original? Do you think it is enough like the original to be acceptable for AM broadcasting?

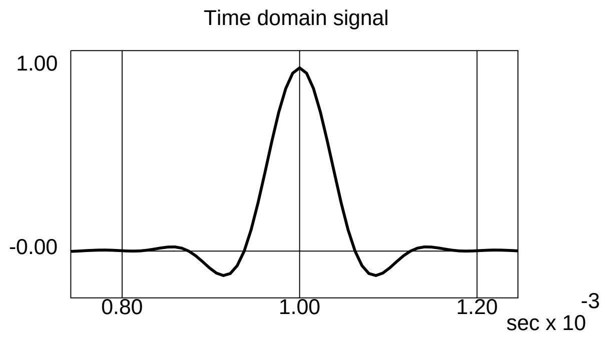
5.5.2 Sampling and multirate

This exercise explores sampling and multirate systems. As with the previous exercise, this one makes extensive use of FFTs to approximate the DTFT in the interval from 0 (d.c.) to 2π (normalized) or the sampling frequency f_s (unnormalized).

1. The first task is to generate an interesting signal that we can operate on. We will begin with the same signal used in the previous exercise, generated by feeding an impulse into the `RaisedCosine` star. Set the parameters of the `RaisedCosine` star as follows:

   length: 256
   symbol_interval: 8
   excessBW: 0.5
   interpolation: 1

   Unlike the previous exercise, you should not leave the *interpolation* parameter on its default value. The time domain should look like the following (after zooming in on the central portion):



   Assume as in the exercise “Modulation” on page 5-70 a sampling frequency of 128kHz. Use the `FFTCx` to compute and plot the magnitude DTFT, properly labeled in absolute frequency. In other words, instead of the normalized sampling frequency 2π, use the actual sampling frequency, 128kHz. Carefully and completely explain in words what would be different about this plot if the signal were a continuous-time signal and the plot of the spectrum were its Fourier transform instead of a DTFT.

2. Subsample the above signal at 64kHz, 32kHz, and 16kHz. To do this, use the `DownSample` star (in the “control” palette) with downsampling factors of 2, 4, and 8. Compare the magnitude spectra. It would be best to plot them on the same plot. To do this, you will need to keep the number of samples consistent in all signal paths. Since the `DownSample` star produces only one sample for every N it consumes, the `FFTCx` star that gets its data should have its *size* parameter proportional to $1/N$ for each path.

Warning: If you fail to make the numbers consistent, you will either get an error message, or your system will run for a very long time. Please be sure you understand synchronous dataflow. Read "Iterations in SDF" on page 5-3**.**

Answer the following questions:

a. Which of the downsampled signals have significant aliasing distortion?

b. What is the smallest sample rate you can achieve with the downsampler without getting aliasing distortion?

3. The next task is to show that sometimes subsampling can be used to demodulate a modulated signal.

   a. First, modulate our "interesting signal" with a *complex exponential* at frequency 32kHz. The complex exponential can be generated using the `expgen` galaxy in the sources palette. Plot the magnitude spectrum, and explain in words how this spectrum is different from the one obtained in the exercise "Modulation" on page 5-70, which modulates with a cosine at 32kHz.

   b. Next, demodulate the signal by downsampling it. What is the appropriate downsampling ratio?

4. The next task is to explore upsampling.

   a. First, generate the signal we will work with by downsampling the original "interesting signal" at a 32kHz sample rate (a factor of 4 downsampling). Then upsample by a factor of 4 using the `UpSample` star. This star will just insert three zero-valued samples for each input sample. Compare the magnitude spectrum of the original "interesting signal" with the one that has been downsampled and then upsampled. Explain in words what you observe.

   b. Instead of upsampling with the `UpSample` star, try using the `Repeat` star. Instead of filling with zeros, this one holds the most recent value. This more closely emulates the behavior of a practical D/A converter. Set the *numTimes* parameter to 4. Compare the magnitude spectrum to that of the original signal. Explain in words the difference between the two. Is this a better reconstruction than the zero-fill signal of part (a)?

   c. Use the `Biquad` star with default parameters to filter the output of the `Repeat` star from part (b). Does this improve the signal? Describe in words how the signal still differs from the original.

5.5.3 Exponential sequences, transfer functions, and convolution

This exercise explores rational Z transform transfer functions.

1. Generate an exponential sequence $a^n u(n)$, with $a = 0.9$, and convolve it with a square pulse of width 10. For this problem, use the following brute-force method for generating the exponential sequence. Observe that

$$a^n = e^{ln(a^n)} = e^{n(ln(a))}.$$

   You can use the `Const` star to generate a constant a, feed that constant into the `Log` star, multiply it by the sequence $n \times u(n)$ generated using the `Ramp` star, and feed the result into the `Exp` star. For your display, try the following options to the `XMgraph` star: "-P -nl -bar".

2. A much more elegant way to generate an exponential sequence is to implement a filter with an exponential sequence as its impulse response. Generate the sequence

$$a^n u(n)$$

by feeding an impulse (`Impulse` star) into a first order filter (`IIR` star). Try various values for a, including negative numbers and values that make the filter unstable.

   a. Let $h(n) = a^n u(n)$ and $x(n) = b^n u(n)$, where $a = 0.9$ and $b = 0.5$. Generate these two sequences using the method above, and convolve them using the convolver block. Now find $h(n)*x(n)$ *without* using a convolver block. Print your block diagram, and don't forget to mark the parameter values on it.

   b. Given the Z transform

$$H(z) = \frac{1 - 0.995z^{-1}}{1 - 1.99z^{-1} + z^{-2}},$$

   use Ptolemy to find and print the inverse Z transform $h(n)$. Find the poles and zeros of the transfer function and use them to explain the impulse response you observe.

3. Generate the following sequences:

$$x(n) = (0.95)^n \sin(0.1n)u(n)$$
$$y(n) = (0.95)^n \sin(0.2n)u(n) \quad ,$$

where $u(n)$ is the unit step function. Estimate the peak value of each signal. Note that you can zoom in xgraph by drawing a box around the region of interest.

4. Given the following difference equation:

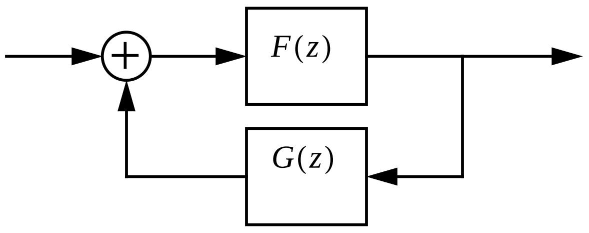
$$y(n) = 2x(n) + 0.75y(n-1) + 0.125y(n-2)$$

find $H(z)$ so that $Y(z) = X(z)H(z)$. Write it down. Use Ptolemy to generate a plot of $h(n)$. Plot $y(n)$ when $x(n)$ is a rectangular pulse of width 5. Assume $y(n) = 0$ for $n < 0$.

5. This problem explores feedback systems. An example of an "all-pole" filter is

$$H(z) = \frac{1}{1 - 2z^{-1} + 1.91z^{-2} - 0.91z^{-3} + 0.205z^{-4}}.$$

Although there are plenty of zeros (at $z = 0$), they don't effect the magnitude frequency response. Hence the name. Although this can be implemented in Ptolemy using the `IIR` star, you are to implement it using only one or more `FIR` star(s) in the standard feedback configuration:



Find $F(z)$ an $G(z)$ to get an overall transfer function of $H(z)$. Then implement it as a feedback system in Ptolemy and plot the impulse response. Is the impulse response infinite in extent?

Note: For a feedback system to be implementable in discrete-time, it must have at least one unit delay (z^{-1}) in the loop. Ptolemy needs for this delay to be explicit, not hidden in the tap values of a filter star. For this reason, you should factor a z^{-1} term out of $G(z)$ and implement it using the delay icon (a small green diamond). Note that the delay is not a star, and is not connected as a star. It just gets placed on top of an arc, as explained in "Using delays" on page 2-47. Also note that Ptolemy requires you to use an explicit `Fork` (in the control palette) if you are going to put a delay on a net with more than one destination.

5.5.4 Linear phase filtering

You can compute the frequency response of a filter in Ptolemy by feeding it an impulse, and connecting the output to an `FFTCx` star. Recall that you will only need to run your system for **one** iteration when you are using an FFT, or you will get several successive FFT computations. The output of the FFT is complex, but may be converted to magnitude and phase using a complex to real (`CxToRect`) followed by a rectangular to polar (`RectToPolar`) converter stars. You can also examine the magnitude in dB by feeding it through the `DB` star before plotting it.

1. Build an FIR filter with real, symmetric tap values. Use any coefficients you like, as long as they are symmetric about a center tap. Look at the phase response. Is it linear, modulo π? Experiment with several sets of tap values, maintaining linear phase. Try long filters and short filters. Experiment with the phase unwrapper star (`Unwrap`), which attempts to remove the 2π ambiguity, keeping continuous phase. Choose your favorite linear-phase filter, and turn in the plots of its frequency response, together a plot of its tap values.
2. For the filter you used in (1), what is the group delay? How is the group delay related to the slope of the phase response?
3. Build an FIR filter with odd-symmetric taps (anti-symmetric). Find the phase response of this filter, and compare it to that in (1). Generate a sine wave (using the `singen` galaxy) and feed it into your filter. What is the phase difference (in radians) between the input cosine and the output? Try different frequencies.
4. Although linear phase is easy to achieve with FIR filters, it can be achieved with other filters using signal reversal. If you run the same signal forwards and backwards through the same filter, you can get linear phase. Given an input $x(n)$ and a filter $h(n)$, compute the output $y(n)$ as follows:

$$g(n) = h(n)*x(n)$$

$$r(n) = h(n)*g(-n)$$

$$y(n) = r(-n) .$$

Obviously, this operation is not causal. Let $f(n)$ be such that

$$y(n) = f(n)*x(n) .$$

Find $f(n)$ in terms of $h(n)$. If $h(n)$ is causal, will $f(n)$ also be causal? Find the frequency response $F(\omega)$, and express it in terms of $|H(\omega)|$ and $\angle H(\omega)$. It will help if you assume all signals are real.

5. All signals in Ptolemy start at time zero, so it is impossible to generate the signal $g(-n)$ used above. However, you can collect a block of samples and reverse them, getting $g(N-n)$, using the `Reverse` star. This introduces an extra delay of N samples. Use a first-order IIR filter (with an exponentially decaying impulse response) to implement $h(n)$. First verify that the above methodology yields an impulse response that is symmetric in time. Then measure the phase response. You can use Ptolemy to adjust the computed phase output to remove the effect of the large delay offset (the center of your symmetric pulse is nowhere near zero). Compare your result against the theoretical prediction in (4).

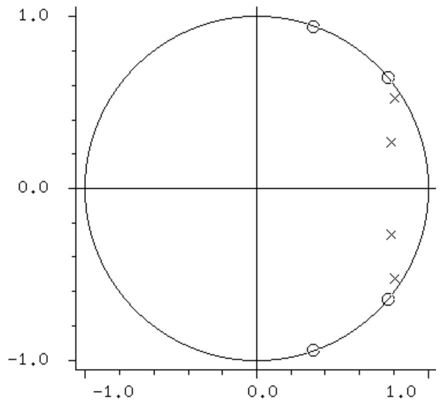
   Hint: You will want the block size of the `Reverse` star to match that used for the `FFTCx` star. Then just run the system through one iteration. Also, you should delay your impulse into the first filter by half the block size. This will ensure a symmetric impulse response, which is what you want for linear phase. The center of symmetry should be half the block size.

5.5.5 Coefficient quantization

1. You will experiment with the following transfer function:

$$H(z) = \frac{1 - 2.1872z^{-1} + 3.0055z^{-2} - 2.1872z^{-3} + z^{-4}}{1 - 3.1912z^{-1} + 4.1697z^{-2} - 2.5854z^{-3} + 0.6443z^{-4}},$$

   which has the following pole-zero plot:



   This is a fourth order elliptic filter.

   a. Implement this filter in the canonical direct form, or direct form II (using the `IIR` star). Plot the magnitude frequency response in dB, and verify that it is what you expect from the pole-zero plot.

   b. The transfer function can be factored as follows, where the poles nearest the unit circle and the zeros close to those poles appear in the second term:

$$H(z) = \left(\frac{1 - 0.6511z^{-1} + z^{-2}}{1 - 1.5684z^{-1} + 0.6879z^{-2}}\right)\left(\frac{1 - 1.5321z^{-1} + z^{-2}}{1 - 1.6233z^{-1} + 0.9366z^{-2}}\right).$$

Implement this as a cascade of two second order sections (using two `IIR` stars). Verify that the frequency response is the same as in part (a). Does the order of the two second order sections affect the magnitude frequency response?

2. You will now quantize the coefficients for implementation in two's complement digital hardware. Assume in all cases that you will use enough bits to the left of the binary point to represent the integer part of the coefficients perfectly. The left-most bit is the most significant bit. You will only vary the number of bits to the right of the binary point, which represent the fractional part. With zero bits to the right of the binary point, you can only represent integers. With one bit, you can represent fractional parts that are either .0 or .5. Other possibilities are given in the table below:

number of bits right of the binary point	**possible values for the fractional part**
2	.0, .25, .5, .75
3	.0, .125, .25, .375, .5, .625, .75, .875
4	.0, .0625, .125, .1875 .25, .3125, .375, ...

You can use the `IIRFix` star to implement this. First, we will study the effects of coefficient quantization only. To minimize the impact of fixed-point internal computations in the `IIRFix` star, set the *InputPrecision, AccumulationPrecision,* and *OutputPrecision* to 16.16 (meaning 16 bits to the right and 16 bits to the left of the binary point) getting more than adequate precision.

   a. For the cascaded second-order sections of problem 1a, quantize the coefficients with two bits to the right of the binary point. Compare the resulting frequency response to the original. What has happened to the pole closest to the unit circle? Do you still have a fourth-order system? Does the order of the second order sections matter now?

   b. Repeat part (a), but using four bits to the right of the binary point. Does this look like it adequately implements the intended filter?

3. Direct form implementations of filters with order higher than two are especially subject to coefficient quantization errors. In particular, poles may move so much when coefficients are quantized that they move outside the unit circle, rendering the implementation unstable. Determine whether the direct form implementation of problem (1a) is stable when the coefficients are quantized. Try 2 bits to the right of the binary point and 4 bits to the right of the binary point. You should plot the impulse response, not the frequency response, to look for instability. How many bits to the right of the binary point do you need to make the system stable?

4. Experiment with the other precision parameters of the `IIRFix` star. Is this filter more sensitive to accumulation precision than to coefficient precision?

5. Many applications require a very narrowband lowpass filter, used to extract the d.c. component of a signal. Unfortunately, the pole locations for second-order direct form 2 structures are especially sensitive to coefficient quantization in the region near $z = 1$. Consequently, they are not very well suited to implementing very narrowband lowpass filters.

   a. The following transfer function is that of a second-order Butterworth lowpass filter:

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.9293z^{-1} + 0.9317z^{-2}} .$$

Find and sketch the pole and zero locations of this filter. Compute and plot the magnitude frequency response. Where is the cutoff frequency (defined to be 3dB below the peak)?

b. Quantize the coefficients to use four bits to the right of the binary point. How many bits to the left of the binary point are required so that all the coefficients can be represented in the same format? Compute and plot the magnitude frequency response of this new filter. Explain why it is so different. What is wrong with it?

c. The following transfer function is a bit better behaved when quantized to four bits to the right of the binary point:

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.7187z^{-1} + 0.7536z^{-2}} .$$

It is also a second order Butterworth filter. Determine where its 3dB cutoff frequency is. Quantize the coefficients to four bits right of the binary point, and determine how closely the resulting filter approximates the original.

d. Use the filter from part (c) (possibly used more than once), together with `Upsample` and `Downsample` stars to implement a lowpass filter with a cutoff of 0.05 radians. Implement both the full precision and quantized versions. Describe qualitatively the effectiveness of this design. Your input and output sample rate should be the same, and the objective is to pass only that part of the input below 0.05 radians to the output unattenuated.

5.5.6 FIR filter design

This lab explores FIR filter design by windowing and by the Parks-McClellan algorithm.

1. Use the `Rect` star to generate rectangular windows of length 8, 16, and 32. Set the amplitude of the windows so that they have the same d.c. content (so that the Fourier transform at zero will be the same).

   a. Find the drop in dB at the peak of the first side-lobe in the frequency domain. Also find the position (in Hz, assuming the sampling interval $T = 1$) of the peak of the first side-lobe. Is the dB drop a function of the length of the window? What about the position?

   b. Find the drop in dB at the side-lobe nearest π radians (the Nyquist frequency) for each of the three window lengths. What relationship would you infer between window length and this drop?

2. Repeat problem 1 with a Hanning window instead of a rectangular window. Be sure to set the *period* parameter of the `Window` star to a negative number in order to get only one instance of the window.

3. An ideal low-pass filter with cutoff at ω_c has impulse response

$$h(n) = \frac{\sin(\omega_c n)}{\pi n} .$$

   This impulse response can be generated for any range $-M \le n \le M$ using the `Raised-Cosine` star from the communications subpalette, or the `Sinc` star from the nonlinear subpalette. This star is actually an FIR filter, so feed it a unit impulse. Its output will be shaped like $h(n)$ if you set the "excess bandwidth" to zero. Set its parameters as follows:

length: 64 (the length of the filter you want)
symbol_interval: 8 (the number of samples to the first zero crossing)
excessBW: 0.0 (this makes the output ideally lowpass).
interpolation: 1

a. What is the theoretical cutoff frequency ω_c given that $h(8) = 0$ is the first zero crossing in the impulse response? Give your answer in Hz, assuming that the sampling interval $T = 1$.

b. Multiply the 64-tap impulse response gotten from the `RaisedCosine` star by Hanning and steep Blackman windows, and plot the original 64-tap impulse response together with the two windowed impulse responses. Which impulse responses end more abruptly on each end?

c. Compute and plot the magnitude frequency response (in dB) of filters with the three impulse responses plotted in part (b). You will want to change the parameter of the `FFTCx` star to get more resolution. You can use an *order* of 9 (which corresponds to a 512 point FFT). You can also set the *size* to 64 since the input has only 64 non-zero samples. Describe qualitatively the difference between the three filters. What is the loss at ω_c compared to d.c.?

4. In this problem, you will use the rather primitive FIR filter design software provided with Ptolemy. The program you will use is called "`optfir`"; it uses the Parks-McClellan algorithm to design equiripple FIR filters. See "optfir — equiripple FIR filter design" on page C-1 for an explanation of how to use it. The main objective in this problem will be to compare equiripple designs to the windowed designs of the previous problem.

   a. Design a 64 tap filter with the passband edge at (1/16)Hz and stopband edge at (0.1)Hz. This corresponds very roughly to the designs in problem 3. Compare the magnitude frequency response to those in problem 3. Describe in words the qualitative differences between them. Which filters are "better"? In what sense?

   b. The filter you designed in part (a) should end up having a slightly wider passband than the designs in problem 3. So to make the comparison fair, we should use a passband edge smaller than (1/16)Hz. Choose a reasonable number to use and repeat your design.

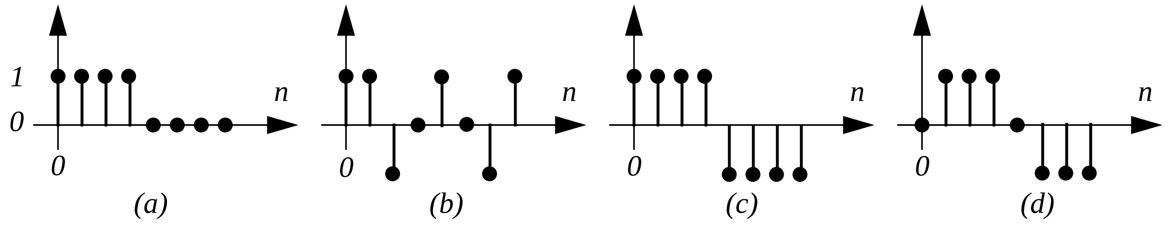
   c. Experiment with different transition band widths. Draw some conclusions about equiripple designs versus windowed designs.

5.5.7 The DFT (discrete Fourier transform)

This exercise explores the DFT, FFT, and circular convolution. Ptolemy has both a `FFTCx` (complex FFT) and a `DTFT` star in the "dsp" palette. The `FFTCx` star has an *order* parameter and a *size* parameter. It consumes *size* input samples and computes the DFT of a periodic signal formed by repeating these samples with period 2^{order}. Only integer powers of two are supported. If $size \le 2^{order}$, then the unspecified samples are given value zero. This can also be viewed as computing samples of the DTFT of a finite input signal of length *size*, padded with $2^{order} - size$ zeros. These samples are evenly spaced from d.c. to 2π, with spacing $2\pi/N$, where $N = 2^{order}$.

The `DTFT` star, by contrast, computes samples of the DTFT of a finite input signal at *arbitrary* frequencies (the frequencies are supplied at a second input port). If you are interested in computing even spaced samples of the DTFT in the whole range from d.c. to the sampling frequency, the `DTFT` star would be far less efficient than the `FFTCx` star. However, if you are interested in only a few samples of the DTFT, then the `DTFT` star is more efficient. For this exercise, you should use the `FFTCx` star.

1. Find the 8 point DFT (*order* = 3, *size* = 8) of each of the following signals:



   Plot the magnitude, real, and imaginary parts on the same plot. Ignoring any slight round-off error in the computer, which of the DFTs is purely real? Purely imaginary? Why? Give a careful and complete explanation. **Hint:** Do not rely on implicit type conversions, which are tricky to use. Instead, explicitly use the `CxToReal` and `RectToPolar` stars to get the desired plots.

2. Let

$$x(n) = \begin{cases} 1, & \text{if } n = 0, 1, 2, \text{or } 3 \\ 0, & \text{otherwise} \end{cases}$$

   as in (a) above. Compute the 4, 8, 16, 32, and 64 point DFT using the `FFTCx` star. Plot the 64 point DFT. Explain why the 4 point DFT is as it is, and explain why the progression does what it does as the order of the DFT increases.

3. Assuming a sample rate of 1 Hz, compare the 128 point FFT (*order* = 7, *size* = 128) of a 0.125 Hz cosine wave to the 128 point FFT of a 0.123 Hz cosine wave. It is easy to observe the differences in the magnitude, so you should plot only the magnitude of the output of the `FFTCx` star. Explain why the DFTs are so different.

4. For the same 0.125 Hz signal of problem 3, compute a DFT of order 512 using only 128 samples, padded by zeros (*order* = 9, *size* = 128; the zero padding will occur automatically). Explain the difference in the magnitude frequency response from that observed in problem 3. Do the same for the 0.123 Hz signal. Is its magnitude DFT much different from that of the 0.125 Hz cosine? Why or why not?

5. Form a rectangular pulse of width 128 and plot its magnitude DFT using a 512 point FFT (*order* = 9, *size* =512). How is this plot related to those in problem 4? Multiply this pulse by 512 samples of a 0.125 Hz cosine wave and plot the 512 point DFT. How is this related to the plot in problem 4? Explain. **Reminder:** If you get an error message "unresolvable type conflict" then you are probably connecting a float signal to both a float input and a complex input. You can use explicit type conversion stars to correct the problem.

6. To study circular convolution, let

$$y(n) = \begin{cases} 1, & \text{if } n = 1, 2, 3, 4, 5, \text{ or } 6 \\ 0, & \text{otherwise} \end{cases}$$

and let $x(n)$ be as given in problem 2. Use the `FFTCx` star to compute the 8 point circular convolution of these two signals. Which points are affected by the overlap caused by circular convolution? Compute the 16 point circular convolution and compare.

5.5.8 Whitening filters

This exercise, and all the remaining ones in this chapter, involve random signals.

1. Implement a filter with two zeros, located at $z = a \pm ja$, where $a = 0.8$, one pole at $z = 0$, and one pole at $z = 0.9$. You may use the `Biquad` or `IIR` star in the "dsp" palette. Filter white noise with it to generate an ARMA process. Then design a whitening filter that converts the ARMA process back into white noise. Demonstrate that your system does what is desired by whatever means seems most appropriate.
2. Implement a causal FIR filter with two zeros at $z = a$ and $z = a^*$, where

$$a = 0.9e^{j\pi/4}.$$

Plot its magnitude frequency response and phase response, using the `Unwrap` star to remove discontinuities in the phase response. Then implement a second filter with two zeros at $1/a$ and $1/a^*$. Adjust the gain of this filter so that it is the same at d.c. as the first filter. Verify that the magnitude frequency responses are the same. Compare the phases. Which is minimum phase? Then implement an allpass filter which when cascaded with the first filter yields the second. Plot its magnitude and phase frequency response.

5.5.9 Wiener filtering

1. Generate an AR (auto-regressive) process $x(n)$ by filtering white Gaussian noise with the following filter:

$$G(z) = \frac{1}{1 - 2z^{-1} + 1.91z^{-2} - 0.91z^{-3} + 0.205z^{-4}}.$$

You can implement this with the `IIR` filter star. The parameters of the star are:

gain: A float: g
numerator: A list of floats separated by spaces: $a_0 \ a_1 \ a_2$ ...
denominator: A list of floats separated by spaces: $b_0 \ b_1 \ b_2$ ...

where the transfer function is:

$$H(z) = g\left(\frac{a_0 + a_1z^{-1} + a_2z^{-2} + \ldots}{b_0 + b_1z^{-1} + b_2z^{-2} + \ldots}\right).$$

More interestingly, you can implement the filter with an FIR filter in the feedback loop. Try it both ways, but turn in the latter implementation.

2. Define the "desired" signal to be

$$d(n) = g(n) * x(n) + w(n) \quad ,$$

where $w(n)$ is a white Gaussian noise process with variance 0.5, uncorrelated with $x(n)$, and $g(n)$ is the impulse response of a filter with the following transfer function:

$$G(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} .$$

Generate $d(n)$.

3. Design a Wiener filter for estimating $d(n)$ from $x(n)$. Verify that the power of the error signal $e(n) = d(n) - y(n)$ is equal to the power of the additive white noise $w(n)$.
4. Use an adaptive LMS filter to perform the same function as the fixed Wiener filter in part 3. Use the default initial tap values for the `LMS` filter star. Compare the error signals for the adaptive system to the error signal for fixed system by comparing their power. How closely does the LMS filter performance approximate that of the fixed Wiener filter? How does its performance depend on the adaptation step size? How quickly does it converge? How much do its final tap value look like the optimal Wiener filter solution?

Ptolemy Hint: The `powerEst` galaxy (in the nonlinear palette) is convenient for estimating power. For the `LMS` star, to examine the final tap values, set the *saveTapsFile* parameter to some filename. This file will appear in your home directory (even if you started pigi in some other directory). To examine this file, just type “`pxgraph -P filename`” in any shell window. The -P option causes each point to shown with a dot. You may also wish to experiment with the `LMSTkPlot` star to get animated displays of the filter taps as they adapt.

5.5.10 Adaptive equalization

1. Generate random sequence of ± 1 using the `IIDUniform` and `Sgn` stars. This represents a random sequence of bits to be transmitted over a channel. Filter this sequence with the following filter (the same filter used in “Wiener filtering” on page 5-81):

$$A(z) = \frac{1}{1 - 2z^{-1} + 1.91z^{-2} - 0.91z^{-3} + 0.205z^{-4}} .$$

Assume this filter represents a channel. Observe that it is very difficult to tell from the channel output directly what bits were transmitted. Filter the channel output with an LMS adaptive filter. Try two mechanisms for generating the error used to update the LMS filter taps:

a. Subtract the LMS filter output from the transmitted bits directly. These bits may be available at a receiver during a start-up, or “training” phase, when a known sequence is transmitted.
b. Use the `Sgn` star to make decisions from the LMS filter output, and subtract the filter output from these decisions. This is a decision-directed structure, which does not assume that the transmitted bits are known at the receiver.

To get convergence in reasonable time, it may be necessary to initialize the taps of the LMS filter with something reasonably close to the inverse of the channel response. Try initializing each tap to the integer nearest the optimal tap value. Experiment with other initial tap values. Does the decision-directed structure have more difficulty adapting than the “training” structure that uses the actual transmitted bits? You may wish to experiment with the `LMSTkPlot` block to get animated displays of the filter taps.

2. For the this problem, you should generate an AR process by filtering Gaussian white noise with the following filter:

$$A(z) = \frac{1}{1 - 1.94z^{-1} + 0.98z^{-2}}.$$

   Construct an optimal one-step forward linear predictor for this process using the `FIR` star, and a similar adaptive linear predictor using the `LMS` star. Display the two predictions and the original process on the same plot. Estimate the power of the prediction errors and the power of the original process. Estimate the prediction gain (in dB) for each predictor. For each predictor, how many fewer bits would be required to encode the prediction error power vs. the original signal with the same quantization error? Assume the number of bits required for each signal to have the same quantization error is determined by the 4σ rule, which means that full scale is equal to four standard deviations.

3. Modify the AR process so that is generated with the following filter:

$$A(z) = \frac{1}{1 - 1.2z^{-1} + 0.6z^{-2}}.$$

   Again estimate the prediction gain in both dB and bits. Explain clearly why the prediction gain is so much lower.

4. In the file `$PTOLEMY/src/domains/sdf/demo/speech.lin` there are samples from two seconds of speech sampled at 8kHz. You need not use all 16,000 samples. The samples are integer-valued with a peak of around 20,000. You may want to scale the signal down. Use your one-step forward linear predictor with the LMS algorithm to compute the prediction error signal. Measure the prediction gain in dB, and note that it varies widely for different speech segments. Identify the segments where the prediction gain is greatest, and explain why. Identify the segments where the prediction gain is small and explain why it is so. Make an engineering decision about the number of bits that can be saved by this coder without appreciable degradation in signal quality. You can read the file using the `WaveForm` star.

5.5.11 ADPCM speech coding

For the same speech file you used in the last assignment, `$PTOLEMY/src/domains/sdf/demo/speech.lin`, you are to construct an adaptive differential pulse code modulation (ADPCM) coder using the “feedback around quantizer” structure and an LMS filter to form the approximate linear prediction. Be sure to connect your LMS filter so that at the receiver, if there are no transmission errors, an LMS filter can also be used in a feedback path, and the LMS filter will exactly track the one in the transmitter. You will use various amounts of quantization.

To assess the ADPCM system, reconstruct the speech signal from the quantized residual, subtract this from the original signal, and measure the noise power. If you have a workstation with a speaker available, listen to the sound, and compare against the original.

1. In your first experiment, do not quantize the signal. Find a good step size, verify that the feedback around quantizer structure works, measure the reconstruction error power and prediction gain. Does your reconstruction error make sense? Compare your prediction gain

result against that obtained in the previous lab. It should be identical, since all you have changed is to use the feedback-around-quantizer structure, but you are not yet using a quantizer.

Assume you have a communication channel where you can transmit N bits per sample. You will now measure the signal quality you can achieve with ADPCM compared to simple PCM (pulse code modulation) over the same channel. In PCM, you directly quantize the speech signal to 2^N levels, whereas in ADPCM, you quantize the prediction error to 2^N levels. For a given N, you should choose the quantization levels carefully. In particular, the quantization levels for the ADPCM case should not be the same as those for the PCM case. Given a particular prediction gain G, what should the relationship be? You should use the `Quant` star to accomplish the quantization in both cases. A useful way to set the parameters of the `Quant` star is as follows (shown for $N = 2$ bits, meaning 4 quantization levels):

thresholds: (-1*s) (0) (1*s)

levels: (-1.5*s) (-0.5*s) (0.5*s) (1.5*s)

where “s” is a universe parameter. This way, you can easily experiment with various quantization spacings without having to continually retype long sequences of numbers.

For each N, you should compare (a) the ADPCM encoded speech signal and (b) the PCM encoded speech signal to the original speech signal. You should make this comparison by measuring the power in the differences between the reconstructed signals and the original. How does this difference compare to the prediction gain?

2. Use $N = 3$ bits.
3. Use $N = 2$ bits.

5.5.12 Spectral estimation

In the Ptolemy “dsp” palette there are three galaxies that perform three different spectral estimation techniques. These are the (1) periodogram, (2) autocorrelation method using the Levinson-Durbin algorithm, and (3) Burg’s method. The latter two compute linear predictor coefficients, and then use these to determine the frequency response of a whitening filter for the random process. The magnitude squared of this frequency response is inverted to get an estimate of the power spectrum of the random process. Study these and make sure you understand how they work. You are going to use all three to construct power spectral estimates of various signals and compare them. In particular, note how many input samples are consumed and produced. If you display all three spectral estimates on the same plot, then you must generate the same number of samples for each estimate. You will begin using only the Burg galaxy.

1. In this problem, we study the performance of Burg’s algorithm for a simple signal: a sinusoid in noise. First, generate a sinusoid with period equal to 25 samples. Add Gaussian white noise to get an SNR of 10 dB.
   a. Using 100 observations, estimate the power spectrum using order 3, 4, 6, and 12th order AR models. You need not turn in all plots, but please comment on the differences.
   b. Fix the order at 6, and construct plots of the power spectrum for SNR of 0, 10, 20, and 30 dB. Again comment on the differences.

c. When the AR model order is large relative to the number of data samples observed, an AR spectral estimate tends to exhibit spurious peaks. Use only 25 input samples, and experiment with various model orders in the vicinity of 16. Experiment with various signal to noise ratios. Does noise enhance or suppress the spurious peaks?

d. Spectral line splitting is a well-known artifact of Burg's method spectral estimates. Specifically, a single sinusoid may appear as two closely spaced sinusoids. For the same sinusoid, with an SNR of 30dB, use only 20 observations of the signal and a model order of 15. For this problem, you will find that the spectral estimate depends heavily on the starting phase of the sinusoid. Plot the estimate for starting phases of 0, 45, 90, and 135 degrees of a cosine wave.

2. In this problem, we study a synthetic signal that roughly models both voiced and unvoiced speech.

   a. First construct a signal consisting of white noise filtered by the transfer function

$$H(z) = \frac{1}{1 - 1.2z^{-1} + 0.6z^{-2}} .$$

   Then estimate its power spectrum using three methods, a periodogram, the autocorrelation method, and Burg's method. Use 256 samples of the signal in all three cases, and order-8 estimates for the autocorrelation and Burg's methods. Increase and decrease the number of inputs that you read. Does the periodogram estimate improve? Do the other estimates improve? How should you measure the quality of the estimates? What order would work better than 8 for this estimate?

   b. Instead of exciting the filter $H(z)$ with white noise, excite it with an impulse stream with period 20 samples. Repeat the spectral estimate experiments. Which estimate is best? Does increasing the number of input samples observed help any of the estimates? With the number of input samples observed fixed at 256, try increasing the order of the autocorrelation and Burg's estimates. What is the best order for this particular signal? Note that deciding on an order for such estimates is a difficult problem.

   c. Voiced speech is often modeled by an impulse stream into an all-pole filter. Unvoiced speech is often modeled by white noise into an all-pole filter. A reasonable model includes some of both, with more noise if the speech is unvoiced, and less if it is voiced. Mix noise and the periodic impulse stream at the input to the filter $H(z)$ in various ratios and repeat the experiment. Does the noise improve the autocorrelation and Burg estimates, compared to estimates based on pure impulsive excitation? You should be able to get excellent estimates using both the autocorrelation and Burg's methods. You may wish to run some of these experiments with 1024 input samples.

5.5.13 Lattice filters

In the Ptolemy "dsp" palette there are four lattice filter stars called: `Lattice`, `RLattice`, `BlockLattice`, and `BlockRLattice`. The "R" refers to "Recursive", so the "`RLattice`" stars are inverse filters (IIR), while the "`Lattice`" stars are prediction-error filters (FIR). The "Block" modifier allows you to connect the `LevDur` or `Burg` stars to the Lattice filters to provide the coefficients. A block of samples is processed with a given set of coefficients, and then new coefficients can be loaded.

1. Consider an FIR lattice filter with the following values for the reflection coefficients: 0.986959, -0.945207, 0.741774, -0.236531.
   a. Is the inverse of this filter stable?
   b. Let the transfer function of the FIR lattice filter be written

   $$H(z) = 1 + h_1 z^{-1} + h_2 z^{-2} + ... + h_M z^{-M}$$

   Use the Levinson-Durbin algorithm to find $h_1, ..., h_M$. Experiment with various methods to estimate the autocorrelation. Turn in your estimates of $h_1, ..., h_M$.
   c. Use Ptolemy to verify that an FIR filter with your computed tap values 1, $h_1, ..., h_M$ has the same transfer function as the lattice filter.
2. In this problem, we compare the biased and unbiased autocorrelation estimates for troublesome sequences.
   a. Construct a sine wave with a period of 40 samples. Use 64 samples into the `Autocor` star to estimate its autocorrelation using both the biased and unbiased estimate. Which estimate looks more reasonable?
   b. Feed the two autocorrelation estimates into the `LevDur` star to estimate predictor coefficients for various prediction orders. Increase the order until you get predictor coefficients that would lead to an unstable synthesis filter. Do you get unstable filters for both biased and unbiased autocorrelation estimates?
   c. Add white noise to the sine wave. Does this help stabilize the synthesis filter?
   d. Load your reflection coefficients into the `BlockLattice` star and compute the prediction error both the biased and unbiased autocorrelation estimate. Which is a better predictor?

Chapter 6. HOF Domain

Authors: *Edward A. Lee*

Other Contributors: *Wan-Teh Chang*
Christopher Hylands
Tom Lane
Alan Kamas
Karim Khiar
Thomas M. Parks

6.1 Introduction

A function is *higher-order* if it takes a function as an argument and/or returns a function. A classic example is *mapcar* in Lisp, which takes two arguments, a function and a list. Its behavior is to apply the function to each element of the list and to return a list of the results. The HOF domain implements a similar function, in the form of a star called `Map`, that can apply any other star (or galaxy) to the sequence(s) at its inputs. Many other useful higher-order functions are also provided by this domain.

The HOF domain provides a collection of stars designed to be usable in all other Ptolemy domains. To preserve this generality, not all interesting higher-order functions can be implemented in this domain. As a consequence, some individual domains may also define higher-order functions. In fact, any higher-order function with domain-specific behavior *must* be implemented in its respective domain. The HOF domain is included as a subdomain by all other domains. In Ptolemy 0.7 and later, HOF can be used in both graphical and non-graphical Ptolemy Tcl interpreters.

A common feature shared by all the stars in this domain is that they perform all of their operations in the `preinitialize` method. Moreover, their basic operation is always to disconnect themselves from the graph in which they appear and then to self-destruct. Since the preinitialization method of the stars in a universe is invoked before the preinitialization method of the scheduler, the scheduler never sees the HOF stars. They will have self-destructed by the time the scheduler is invoked. This is why these stars will work in any domain. In code generation domains, an important feature of the HOF stars is that they add no run-time overhead at all, since they self-destruct before code generation begins, and therefore do not appear in any form in the generated code.

Many of the HOF stars will replace themselves with one or more instances of another star or galaxy, called the *replacement block*. Replacement blocks generally go into the graph in the same position originally occupied by the HOF star, but different HOF stars will connect these replacement blocks in different ways.

Some HOF stars have no replacement block. Before they self destruct, they will typically only alter the connections in the graph without adding any new blocks. An example is the `BusMerge` block, which merges two busses into one wider bus. These stars are called *bus* *manipulation* stars.

The experienced reader may have some difficulty connecting the concept of higher-order functions, as implemented in this domain, to that used in functional programming. This issue is covered in some depth in [Lee95], but we can nonetheless give a brief motivation here. In functional languages, there is no syntactic difference between a function argument that is a data value, one that is a stream (an infinite sequence of data values), and one that is a function. In visual programming, however, functions typically have two very different syntaxes for their arguments. Ptolemy is no exception. Stars and galaxies in Ptolemy are functions with two kinds of arguments: input streams and parameters. The HOF domain only contains stars where a parameter may be function. It does not contain any stars that will accept functions at their input portholes as part of an input stream, or produce functions at their output portholes. Although in principle such higher-order functions can be designed in Ptolemy, their behavior would not be independent of their domain, so the HOF domain would be the wrong place for them.

6.2 Using the HOF domain

The HOF stars are found in the main palettes of the domains that use them. For example, the HOF stars used in the SDF domain are found in the main SDF palette. Typically, domains that include the HOF stars will also include demos that use those stars in their demo palette. Thus, the DE demo palette contains a section of Higher Order Function demonstrations. The HOF stars can be used just as if they belonged to the domain in which you are working. Although the examples given below are drawn from the SDF domain, please keep in mind this versatility.

6.2.1 The Map star and its variants

The `Map` star is the most basic of all HOF stars. Its icon is shown below:

It has the following parameters:

blockname	The name of the replacement block.
where_defined	The full path and facet name for the definition of blockname.
parameter_map	How to set the parameters of the replacement block.
input_map	How to connect the inputs.
output_map	How to connect the outputs.

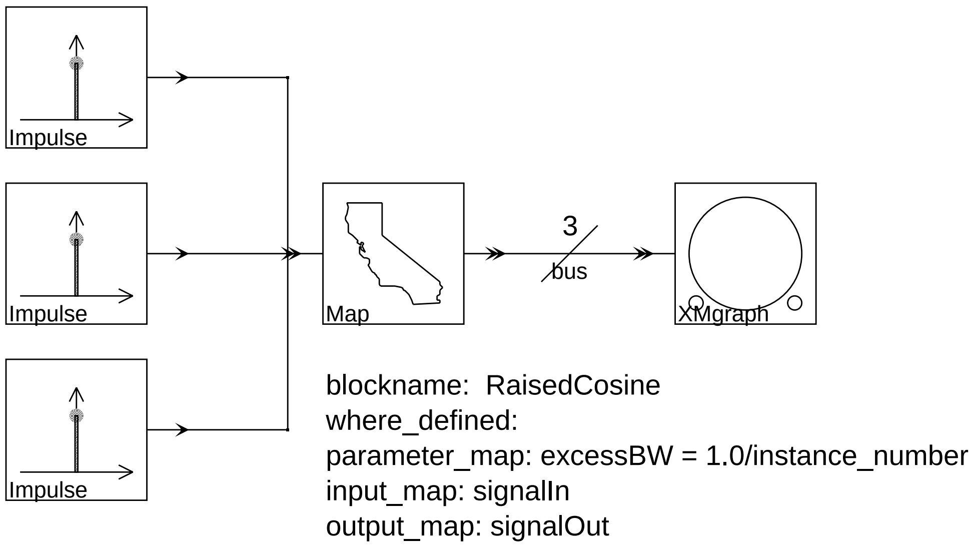
The name of the replacement block is given by the *blockname* parameter. If the replacement block is a galaxy, then the *where_defined* parameter should give the full name (including the full path) of a facet that, when compiled, will define the block. This path name may (and probably should) begin with the environment variable `$PTOLEMY` or ~*username*. This lends a certain immunity to changes in the filesystem organization. Currently, the file specified must be an oct facet, although in the future, other specifications (like `ptcl` files) may be allowed. Usually, the oct facet simply contains the definition of the replacement galaxy. If the replacement block is a built-in star, then there is no need to give a value to the *where_defined* parameter.

The `Map` star replaces itself in the graph with as many instances of the replacement block as needed to satisfy all of the inputs to the Map star. Consider the example shown in figure 6-1. The replacement block is specified to be the built-in `RaisedCosine` star. Since this is built-in, there is no need to specify where it is defined, so the *where_defined* parameter is blank. The `RaisedCosine` star has a single input named *signalIn* and a single output named *signalOut*, so these names are given as the values of the *input_map* and *output_map* parameters. The *parameter_map* parameter specifies the values of the *excessBW* parameter for each instance of the replacement block to be created; *excessBW* specifies the excess bandwidth of the raised cosine pulse generated by the star. The syntax of the *parameter_map* parameter is discussed in detail below, but we can see that the value of the *excessBW* parameter will be 1.0 for the first instance of the `RaisedCosine` star, 0.5 for the second, and 0.33 for the third.

The horizontal slash through the last connection on the right in figure 6-1 is a `Bus`, which is much like a delay in that the icon is placed directly over the arc without any connections. Its single parameter specifies the number of connections that the single wire represents. Here, the bus width has to be three or the `Map` star will issue an error message. This is because there are three inputs to the `Map` star, so three instances of the `RaisedCosine` star will be created. The three outputs from these three instances need somewhere to go. The result of running this system is shown in figure 6-2

The block diagram in figure 6-1 is equivalent to that in figure 6-3. Indeed, once the preinitialization method of the `Map` star has run, the topology of the Ptolemy universe will be exactly as figure 6-3. The `Map` star itself will not appear in the topology, so examining the topology with, for example, the `ptcl print` command will not show a `Map` star instance.

In both figures 6-1 and 6-3, the number of instances of the `RaisedCosine` star is



blockname: RaisedCosine
where_defined:
parameter_map: excessBW = 1.0/instance_number
input_map: signalIn
output_map: signalOut

FIGURE 6-1: An example of the use of the `Map` star to plot three different raised cosine pulses.

specified graphically. In figure 6-1, it is specified by implication, through the number of instances of the `Impulse` star. In figure 6-3 it is specified directly. Neither of these really takes advantage of higher-order functions. The block diagram in figure 6-4 is equivalent to both 6-1 and 6-3, but can be more easily modified to include more or fewer instances of the `RaisedCosine` star. It is only necessary to modify parameters, not the graphical representation. For example, if the value of the bus parameters in figure 6-4 were changed from 3 to 10, the system would then plot ten raised cosines instead of three.

The left-most star in figure 6-4 is a variant of the `Map` star called `Src`. It has no inputs, and is used when the replacement block is a pure source block with no input. (This is a separate star type only for historical reasons; a `Map` icon with zero inputs would work as well.

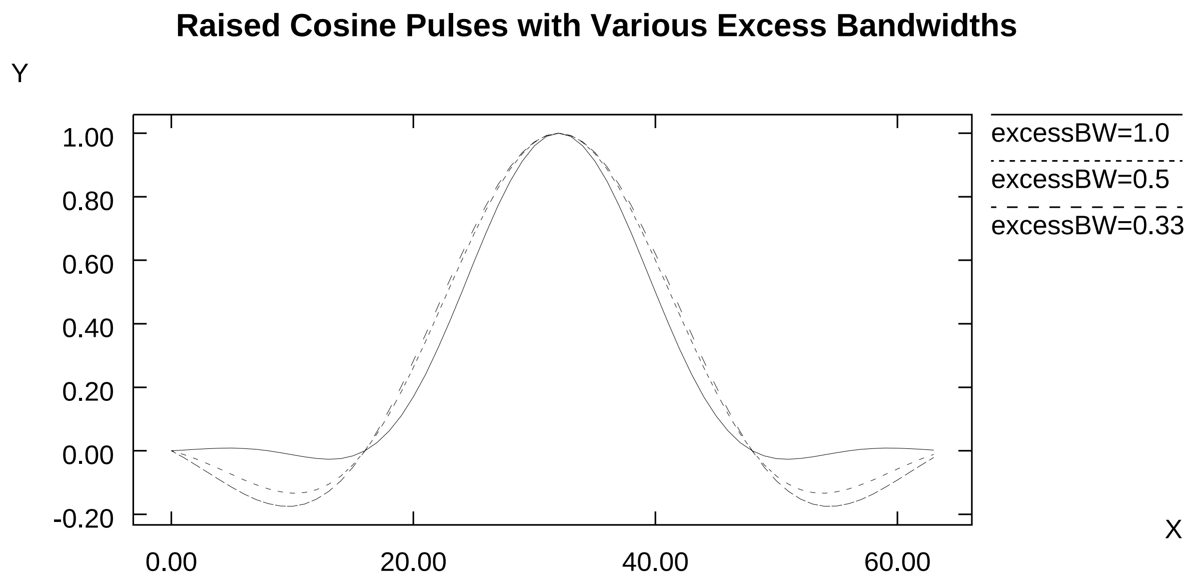


FIGURE 6-2: The plot that results from running the system in figure 6-1.

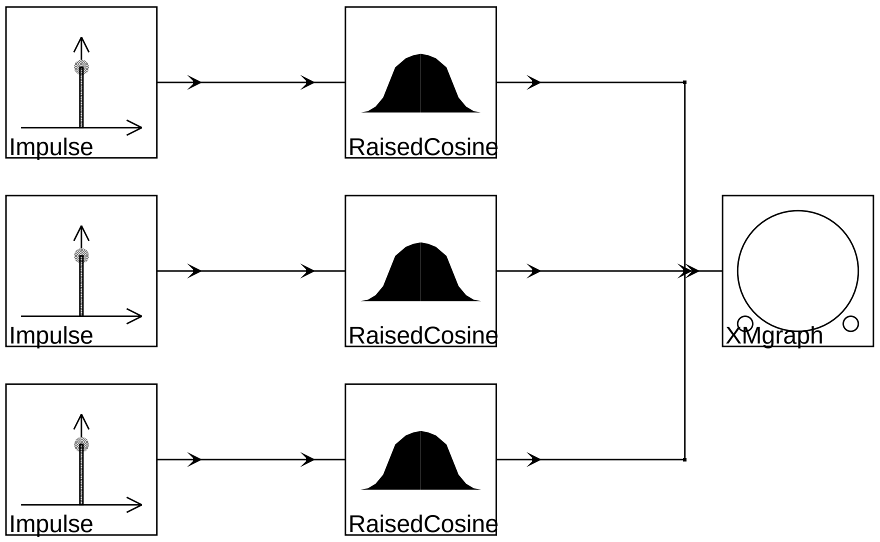
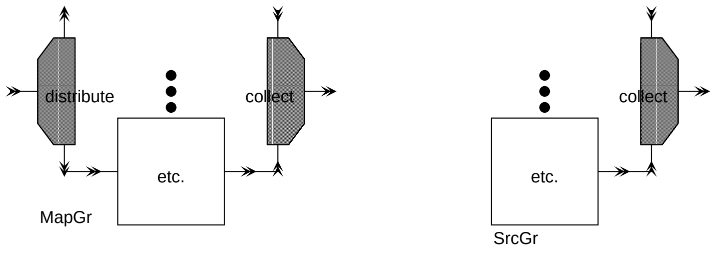


FIGURE 6-3: A block diagram equivalent to that in figure 6-1, but without higher-order functions.

Indeed, for the case of a pure sink replacement block, a Map icon with zero outputs is used.)

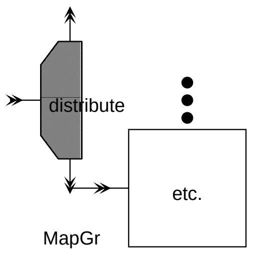
Graphical versions of the Map star

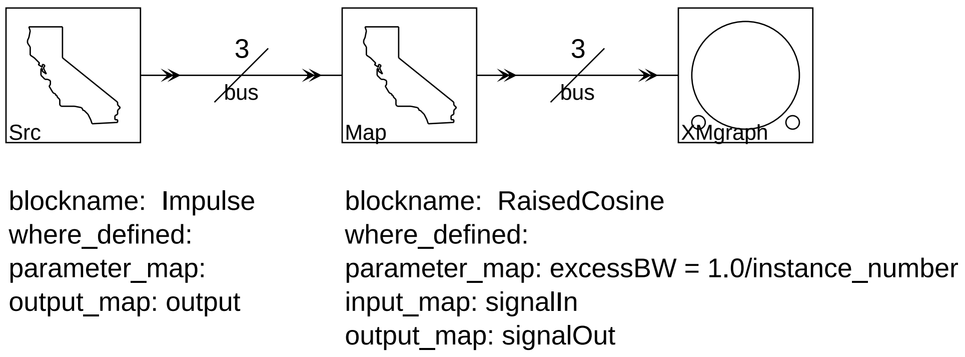
Variants of the `Map` and `Src` stars, called `MapGr` and `SrcGr`, have the following icons:



It is important to realize that `MapGr` and `SrcGr` are single icons, each representing a single star. The complicated shape of the icon is intended to be suggestive of its function when it is found in a block diagram. The `MapGr` and `SrcGr` stars work just like the `Map` and `Src` stars, except that the user specifies the replacement block graphically rather than textually. For example, the system in figure 6-4 can be specified as shown in figure 6-5. Notice that replacement blocks `Impulse` and `RaisedCosine` each have one instance wired into the block diagram as an example. Thus, there is no reason for the *blockname*, *where_defined*, *input_map*, or *output_map* parameters. The `MapGr` and `SrcGr` stars have only a single parameter, called *parameter_map*. The syntax for this parameter is the same as for the `Map` star, and is fully explained below.

A variant of the `MapGr` star has the icon shown below:





blockname: Impulse	blockname: RaisedCosine
where_defined:	where_defined:
parameter_map:	parameter_map: excessBW = 1.0/instance_number
output_map: output	input_map: signalIn
	output_map: signalOut

FIGURE 6-4: A block diagram equivalent to that in figures 6-1 and 6-3, except that the number of instances of the `RaisedCosine` and `Impulse` stars can be specified by a parameter.

This version just represents a `MapGr` star with no outputs.

A more complex application of the `MapGr` star is shown in figure 6-6. Here, the replacement block is a `Commutator`, which can take any number of inputs. The bus connected to its input multiporthole determines how many inputs will be used in each instance created by the `MapGr` star. In the example in figure 6-6, it is set to 2. Thus, each instance of the replacement block processes two input streams and produces one output stream. Consequently, the input bus must be twice as wide as the output bus, or the `MapGr` star will issue an error message. This example produces the plot shown in figure 6-7. A key advantage of higher-order functions becomes apparent when we realize that these parameters can be changed. If the parameters are modified to generate 8 instances of the `Commutator` star, then the output plot will be as shown in figure 6-8.

Setting parameter values

The *parameter_map* parameter of the `Map` star and related stars can be used to set parameter values in the replacement blocks. The *parameter_map* is a string array, a list of

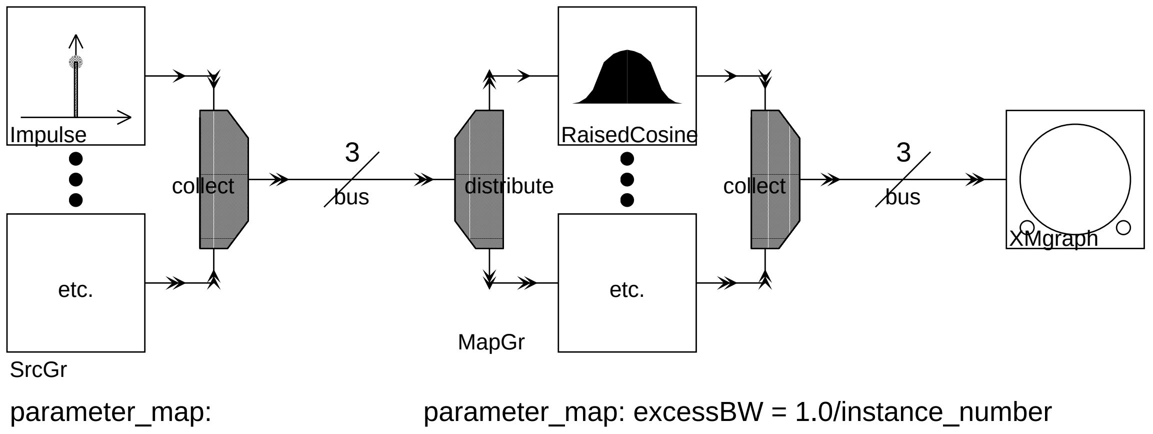


FIGURE 6-5: A block diagram equivalent to that in figure 6-4 except that the replacement blocks for the two higher-order stars are specified graphically rather than textually.

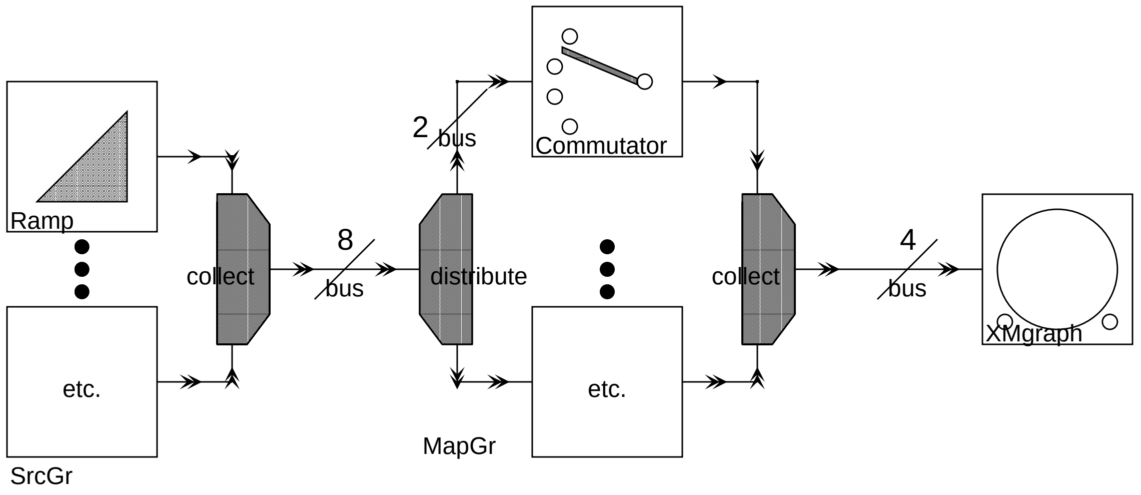


FIGURE 6-6: A more complicated example using higher-order functions.

strings. The strings are in pairs, where the pairs are separated by spaces, and there are four acceptable forms for each pair:

```
        name value
        name(number) value
        name = value
        name(number) = value
```

There should be no spaces between *name* and (*number*), and the name cannot contain spaces, =, or `(`. In all cases, *name* is the name of a parameter in the replacement block. In the first and third cases, the value is applied to all instances of the replacement block. In the second and fourth cases, it is applied only to the instance specified by the instance *number*, (which starts with 1). The third and fourth cases just introduce an optional equal sign, for readability. If the = is used, there must be spaces around it.

The *value* can be any usual Ptolemy expression for giving the value of a parameter. If this expression has spaces in it, however, then the value should appear in quotation marks so that the whole expression is kept together. If the string `instance_number` appears anywhere

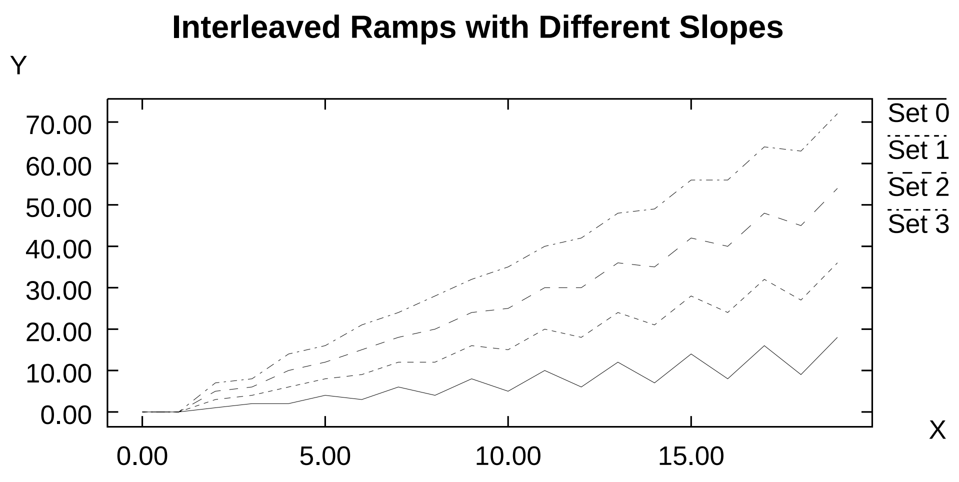


FIGURE 6-7: The plot created by running the system in figure 6-6.

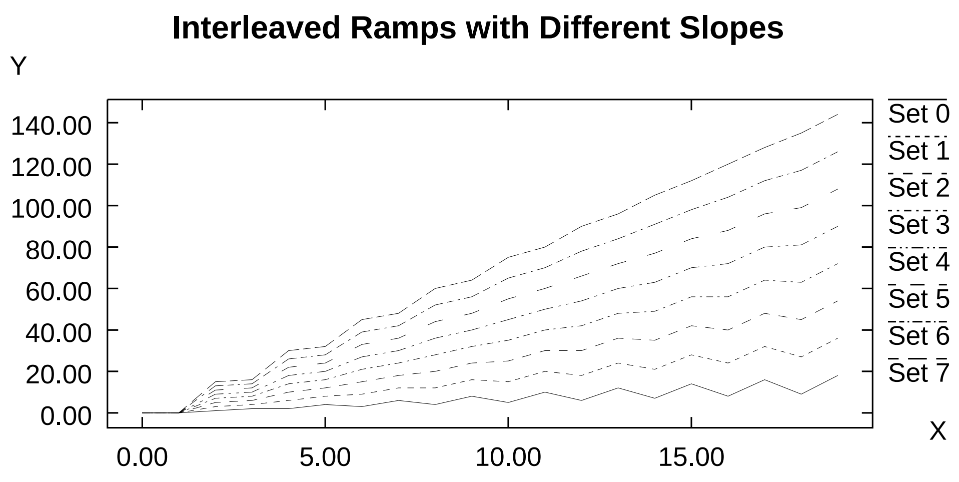


FIGURE 6-8: If the parameters in figure 6-6 are modified to double the number of plots, we get this output.

in *value*, it will be replaced with the instance number of the replacement block. Note that it need not be a separate token. For example, the value `xxxinstance_numberyyy` will become `xxx1yyy` for the first instance, `xxx2yyy` for the second, etc. After all appearances of the string `instance_number` have been replaced, *value* is evaluated using the usual Ptolemy expression evaluator for initializing String Array states.

For example, in figure 6-1, the `Map` star has a *blockname* of `RaisedCosine`, and a *parameter_map* of

```
excessBW = 1.0/instance_number
```

When the system is run, the `Map` star will create three instances of RaisedCosine. The first instance will have its excessBW parameter set to 1.0 (which is 1/1), the second instance of RaisedCosine will have a excessBW of 0.5 (1/2), and the third will have a excessBW of 0.33 (1/3). Since the other RaisedCosine parameters are not mentioned in the parameter_map, they are set to their default values.

As a further example, suppose *parameter_map* of the `Map` star in figure 6-1 were set to

```
excessBW(1) 0.6 excessBW(2) 0.5 excessBW(3) 0.4 length 128
```

The first `RaisedCosine` would then have an *excessBW* of 0.6, the second would have an *excessBW* of 0.5 and the third would have 0.4 for its *excessBW*. All three of the `Raised-Cosine` stars would have a *length* of 128 instead of the default length.

Number of replacement blocks

The number of instances of the replacement block is determined by the number of input or output connections that have been made to the `Map` star. Suppose the `Map` star has M_I inputs and M_O outputs connected to it. Suppose further that the replacement block has B_I input ports and B_O output ports. Then

$$N = \frac{M_I}{B_I} = \frac{M_O}{B_O}$$

is the number of instances that will be created. This must be an integer. Moreover, the number of input and output connections must be compatible (must satisfy the above equality), or you will get an error message like: “too many inputs for the number of outputs.”

How the inputs and outputs are connected

The first B_I inputs to the `Map` star will be connected to the inputs of the first instance of the replacement block. To determine in what order these B_I connections should be made, the names of the inputs to the replacement block should be listed in the *input_map* parameter in the order in which they should be connected. There should be exactly B_I names in the *input_map* list. The next B_I inputs to the `Map` star will be connected to the next replacement block, again using the ordering specified in *input_map*. Similarly for the outputs. If there are no inputs at all, then the number of instances is determined by the outputs, and vice versa.

For `MapGr` and its variants, there is no *input_map* or *output_map* parameter; all connections are specified graphically. If the replacement block has more than one input or more than one output port, these connections must be grouped into a bus connection to the appropriate port of the `MapGr` star. A `HOFNop` star (see “Bus manipulation stars” on page 6-13) can be inserted between the `MapGr` star and the replacement block to perform this grouping. The order of the connections to the `Nop` star then determines the precise order in which `MapGr` makes connections. In this way the `Nop` star's icon provides the same control graphically that *input_map* and *output_map* do textually. (By the way, this use of `Nop` is the only exception to the normal rule that only a single replacement-block icon can be connected to a `MapGr` star. For both `Map` and `MapGr`, if you want to replicate a multiple-star grouping then you need to create a galaxy representing the group to be replicated. The same is true of the remaining HOF stars that generate multiple instances of a block.)

Substituting blocks with multiple input or output ports

When the replacement block has a multiple input port or a multiple output port (shown graphically as a double arrowhead), the name given in the *input_map* parameter should be the name of the multiple port, repeated for however many instances of the port are desired.

For example, the `Add` star has a multiple input port named "input". If we want the replacement `Add` star(s) to have two inputs each, then *input_map* should be `input input`. If we want three inputs in each replacement block, then input_map should be `input input input`. Note that *input_map* and *output_map* are both of the String Array type. Thus one can use the shortcut string `input[3]` instead of the cumbersome `input input input` string. These two forms are equivalent as `input[3]` is converted automatically to `input input input` when the parameter is initialized by Ptolemy.

For `MapGr` and its variants, the number of connections to a multiporthole of the replacement block is controlled by placing a bus icon on the connection, as was illustrated earlier.

A note about data types

All the HOF stars show their input and output datatypes as ANYTYPE. In reality, the type constraints are those of the replacement blocks, which might have portholes of specific types.

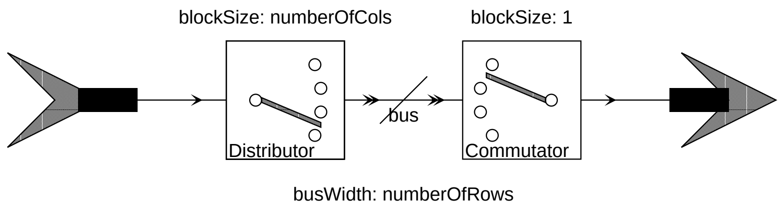
The HOF stars rewire the schematic before any attempt is made to determine porthole types, so the actual assignment of particle types is the same as if the schematic had been written out in full without using any HOF stars.

This was not true in Ptolemy versions prior to 0.7. In prior versions, porthole type assignment occurred before HOF star replacement, which had various unpleasant consequences. For example, the `Map` star used to constrain its input and output particle types to be the same, which interfered with using a replacement block that changed particle types. Also, it was necessary to have numerous variants of the `Src` and `SrcGr` stars, one for each possible output particle type. (If you have any old schematics that contain the type-specific `Src` or `SrcGr` variants, you'll need to use `masters` or `ptfixtree` to replace them with the generic `Src` or `SrcGr` icons.)

6.2.2 Managing multidimensional data

There are many alternatives in Ptolemy for managing multidimensional data. One simple possibility is to use the `MatrixParticle` class. This encapsulates a matrix into a single particle. Another alternative is to use the `Message` class to define your own multidimensional data structure. A third alternative (which is still highly experimental) is to use the multidimensional synchronous dataflow (`MDSDF`) domain. A fourth alternative, discussed here, is to embed your multidimensional data into one-dimensional streams. Higher-order functions become extremely useful in this case. Our discussion will center on using the SDF domain, although the same principles could be applied in other domains as well.

A two-dimensional array of data can be embedded in a one-dimensional stream by rasterizing it. This means that the sequence in the stream consists of the first row first, followed by the second row, followed by the third, etc. This is one example of a *multiprojection,* so called because higher-dimensional data is projected onto a one-dimensional sequence. Typically, however, we wish to perform some operations row-wise, and others column-wise, so the rasterized format can prove inconvenient. Row-wise operations are easy if the data is rasterized, but column-wise operations are awkward. Fortunately, in the SDF domain, we can transpose the data with a cascade of two stars, a `Distributor` and a `Commutator`, as shown below:



If the input is row-wise rasterized, then the output will be column-wise rasterized, meaning that the first column will come out first, then the second column, then the third, etc. It has been shown that any transposition of any arbitrary multiprojection can be accomplished with such a cascade [Khi94].

As an example of the use of a multi-projection transformation, consider the two-dimensional FFT shown in figure 6-9. Recall that a two-dimensional discrete Fourier transform can be implemented by first applying a one-dimensional DFT to the rows and then applying a one-dimensional DFT to the columns. The system in figure 6-9 does exactly this. To see how it works, recall that the `FFTCx` star in the SDF domain has two key parameters, the *order* and the *size*. The *size* is the number of input samples read each time the FFT is computed. For the row FFT, this should be equal to the number of columns. These samples are then padded with zeros (if necessary) to get a total of 2^{order} samples. The `FFTCx` star then computes a 2^{order} point FFT, producing 2^{order} complex outputs. In figure 6-9, these outputs are then transposed so that they are column-wise rasterized. The second `FFTCx` star then computes the FFT of the columns. The output is column-wise rasterized.

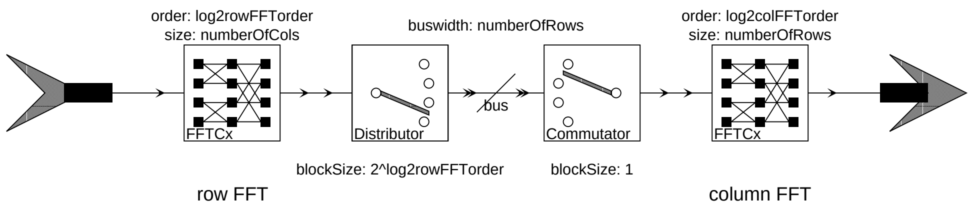


FIGURE 6-9: A two-dimensional FFT operating on rasterized input data and using the `Distributor` and `Commutator` stars to transpose one multiprojection into another.

The effect of a transposition can be accomplished using higher-order functions in a way that is sometimes more intuitive. In particular, a matrix can be represented as a bus, where each connection in the bus carries one row, and the width of the bus is the number of rows. To make this concrete, the same two-dimensional FFT is re-implemented in figure 6-10 using this representation. The rasterized input is first converted into a bus, where each connection in the bus represents one row. The `MapGr` star is then used to apply the FFT to each signal in the bus. The results are recombined in column-wise rasterized format, and the column FFT is computed.

These examples are meant to illustrate the richness of possibilities for manipulating data. A more complete discussion of these issues and their application to radar signal processing are given in [Khi94].

6.2.3 Other higher-order control structures

The `Map` star and its variants apply instances of their replacement block in parallel to the set of input streams. Another alternative is provided by the `Chain` star, which strings together some specified number of instances of the replacement block in series. The parameters are similar to those of the `Map` star, except for the addition of *internal_map*. The *internal_map* parameter specifies connections made between successive instances of the replacement block in the cascade. It should consist of an alternating list of output and input names for the replacement block.

An example of the use of the `Chain` star is a string of biquad filters in series. The `IIR` filter star, which can be used to create a biquad filter, has an input named “signalIn” and an output named “signalOut.” To have a string of these stars in series, one would want the output of the first `IIR` star in the series to be connected to the input of the second star. And the output of the second star should be connected to the input of the third, etc. Thus, a `Chain` star that is a series of biquad filters would have an *internal_map* of

```
signalOut signalIn
```

to specify that the output of one block is connected to the input of the next.

Another variant is the `IfElse` block. This star is just like `Map`, except that it has two possible replacement blocks. If the *condition* parameter is `TRUE`, then the *true_block* is used. Otherwise, the *false_block* is used. It is important to realize that the *condition* parameter is

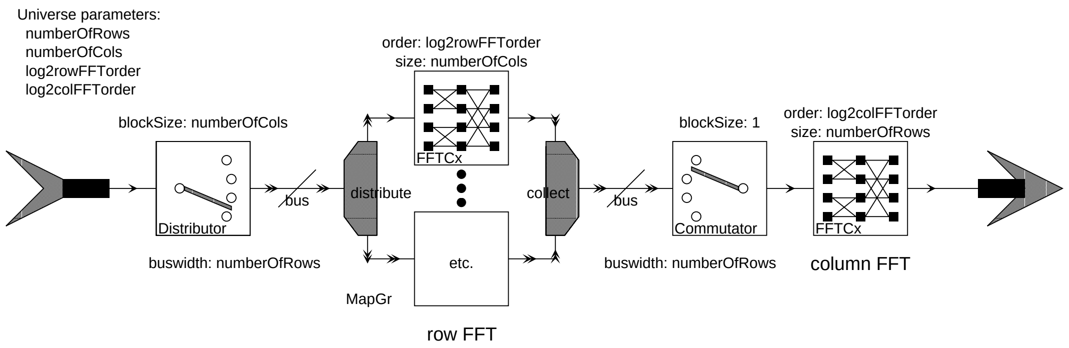


FIGURE 6-10: A two-dimensional FFT implemented using a higher-order function for the row FFTs.

evaluated at preinitialization time. Once a replacement block has been selected, it cannot be changed. There are two uses for this block. It can be used to parameterize a galaxy in such a way that the parameter determines which of two functions is used within the computation. More interestingly, it can be used to implement statically-evaluated recursion.

6.2.4 Statically evaluated recursion

The `Map` star and its variants replace themselves with an instance of the block specified as the replacement block. What if that block is a galaxy within which the very `Map` star in question sits? This is a recursive reference to the galaxy, but a rather awkward one. In fact, in such a configuration, the preinitialization phase of execution will never terminate. The user has to manually abort such an execution in order to get it to terminate.

The `IfElse` star, however, can conditionally specify one of two replacement blocks. The *condition* parameter determines which block. One of the two replacement blocks can be a recursive reference to a galaxy as long as the *condition* parameter is modified. When the condition parameter changes state, going from `TRUE` to `FALSE` or `FALSE` to `TRUE`, then the choice of replacement block inside the new galaxy instance will change. This can be used to terminate the recursion.

Consider the example shown in figure 6-11. This galaxy has a single parameter, *log2framesize*. It will read $2^{log2framesize}$ input particles and rearrange them in bit reversed order. That is, they will emerge from the galaxy as if their binary address had been interpreted with the high-order bit reinterpreted as a low-order bit, and vice versa. Suppose for example that *log2framesize* = 3, and that the input sequence is 10,11,12,13,14,15,16,17. Then the output sequence[1] will be 10,14,12,16,11,15,13,17. To accomplish this, the *bit_reverse* galaxy uses two `IfElse` stars, each with a conditional recursive reference to the *bit_reverse* galaxy.

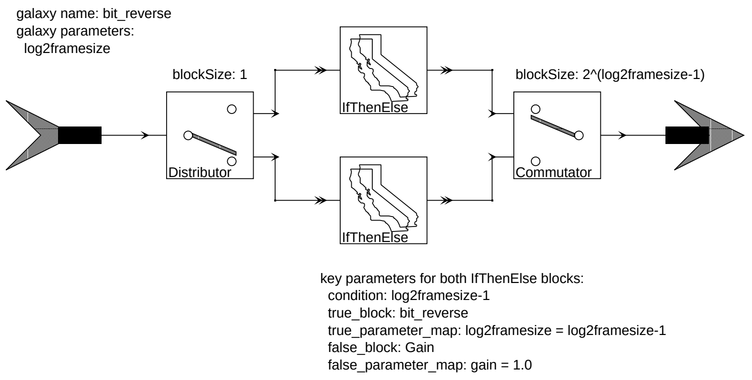

galaxy name: bit_reverse
galaxy parameters:
log2framesize
blockSize: 1
blockSize: 2^(log2framesize-1)
IfThenElse
IfThenElse
Distributor
Commutator
key parameters for both IfThenElse blocks:
condition: log2framesize-1
true_block: bit_reverse
true_parameter_map: log2framesize = log2framesize-1
false_block: Gain
false_parameter_map: gain = 1.0

FIGURE 6-11: A recursive galaxy, where the `IfElse` HOF star replaces itself with an instance of the same galaxy until its *condition* parameter gets to zero.

1. For those unfamiliar with bit-reversed addressing, here is a quick introduction. Since *log2framesize* is 3, galaxy will read in 2^3=8 values at a time. The first value (10) has address 0 (since computers always seem to count from zero) which is 000 in binary. Reversed, its address is still 000 so it is output first. The second value (11) has address 1 which is 001 in binary. Reversed, its address is 100 binary which is 4. Thus the value (11) is output in the fifth spot. As a final example, the seventh value (16) has address 6 which is 110 in binary. Reversed, its binary value is 011 which is 3 and the value (16) is output forth. After the first 8 values are read, the cycle is repeated for the next 8 values.

The condition is *log2framesize*-1, and the *log2framesize* parameter for the inside instances of the galaxy is set to *log2framesize*-1. When *log2framesize* gets to zero, the replacement block becomes a `Gain` star with unity gain (which of course has no effect). This terminates the recursion.

With *log2framesize* = 3, after the preinitialization phase, the topology of the galaxy will have become that shown in figure 6-12. It is much easier to see by inspection of this topology how the bit reversal addressing is accomplished. It is also easier to see how this operation could be made more efficient (the innermost cluster of `Distributors`, `Gains`, and `Commutators` has no effect at all). Unfortunately, we currently have no mechanism for automatically displaying this expanded graph visually. It can, however, be examined using `ptcl`.

The *bit_reverse* galaxy performs the sort of data manipulation that is at the heart of the decimation-in-time FFT algorithm. See [Lee94] for an implementation of that algorithm using these same techniques (or see the demos).

6.2.5 Bus manipulation stars

One consequence of the introduction of higher-order functions into Ptolemy is that busses have suddenly become much more useful than they used to be. Recall that the bus icon resembles a diagonal slash, as shown in figure 6-1, and is placed over a connection, much like a delay. Its single parameter specifies the width of the bus.

Fortunately, while increasing the demand for busses, higher-order functions also provide a cost effective way to manipulate busses. Like the `Map` star and its variants, the bus manipulation stars in the HOF domain modify the graph at preinitialization time and then self-destruct. Thus, they can operate in any domain, and they introduce no run-time overhead.

An example of the use of the `BusSplit` star is shown in figure 6-13. A bank of 12 random number generators produces its output on a bus. The bus is then split into two busses of

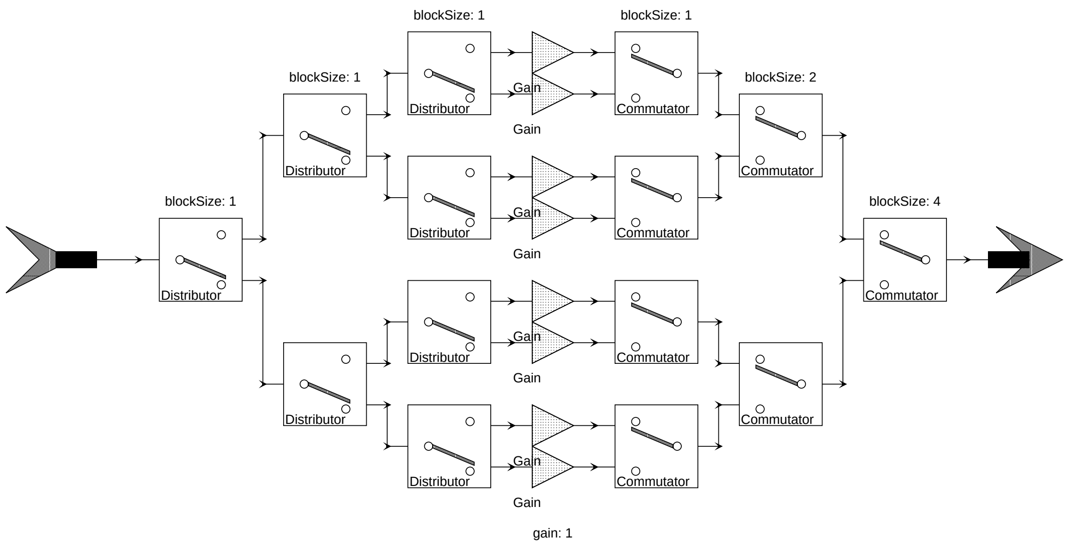
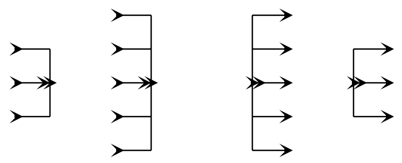


FIGURE 6-12: An expansion of the graph in figure 6-11, representing its topology after the preinitia ization phase assuming *log2blocksize* at the top level is 3.

width 6 so that subsets of the signals can be displayed together. The BusSplit star rewires the graph at preinitialization time and then self-destructs. Thus, it introduces zero run-time overhead.

A more interesting bus manipulation star is the Nop star, so called because it really performs no function at all. It can have any number of inputs, but the number of outputs must be the same as the number of inputs. All it does is connect its inputs to its outputs (at preinitialization time) and the self-destruct. It has many icons, four of which are shown below:



The icon on the left has three individual input ports, and simply combines them into an output multiporthole. This multiporthole would normally be connected to a bus, which must be of width three. Thus, this icon provides a way to create a bus from individual connections. The next icon is similar, except that it has five input lines. The next two icons do the reverse. They are used to break out a bus into its individual components.

Examples of the uses of Nop stars are shown in figure 6-14. Three signals are individually generated at the left by three different source stars. These signals are then combined into a bus of width three using a Nop star. The bus is then broken out into three individual lines, which are fed to three Gain stars. The most interesting use of the Nop star, however, is the one on the right. The XMgraph star shown there has a multiporthole input. The Nop star is simply deposited on top of the multiporthole to provide it with three individual inputs. Why do this? Because when connecting multiple signals to a multiporthole input, as done for example in figure 6-3, it is difficult to control which input line goes to which specific porthole in the multiporthole set. Putting the Nop star on the porthole gives us this control with no additional runtime cost.

Recall that many stars in Ptolemy that have multiportholes have multiple icons, each

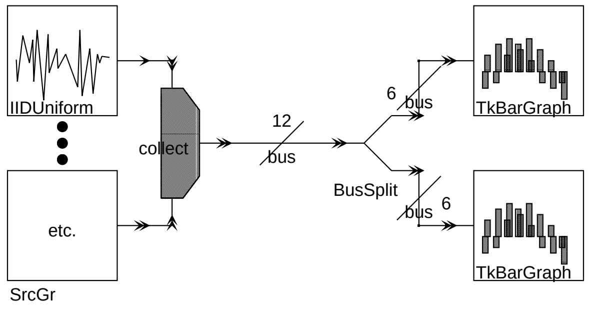


FIGURE 6-13: A BusSplit star is used to divide a set of signals into two subsets for separate display.

icon configured with a different number of individual ports. This proliferation of icons is no longer necessary, and these icons will disappear from the palettes in future versions of Ptolemy. This will considerably reduce clutter in the Ptolemy palettes.

6.3 An overview of the HOF stars

The Higher Order Function stars are accessed through the main palette of the domains that support HOF. For example, the HOF stars are a sub-palette of the SDF star palette since the SDF domain supports HOF. The top-level palette for the HOF domain is shown in figure 6-15.

6.3.1 Bus manipulation stars

The top group in the main HOF palette are the bus manipulation stars, summarized below:

`BusMerge`	Bridge inputs to outputs and then self-destruct. This star merges two input busses into a single bus. If the input bus widths are M_1 and M_2, and the output bus width is N, then we require that $N = M_1 + M_2$. The first M_1 outputs come from the first input bus, while the next M_2 outputs come from the second input bus.
`BusSplit`	Bridge inputs to outputs and then self-destruct. This star splits an input bus into two. If the input bus width is N, and the output bus widths are M_1 and M_2, then we require that $N = M_1 + M_2$. The first M_1 inputs go the first output bus, while the next M_2 inputs go to the second output bus.
`BusInterleave`	Bridge inputs to outputs and then self-destruct. This star interleaves two input busses onto a single bus. The two input busses must have the same width, which must be half the width of the output bus. The input signals are connected to the output in an alternating fashion.
`BusDeinterleave`	Bridge inputs to outputs and then self-destruct. This star

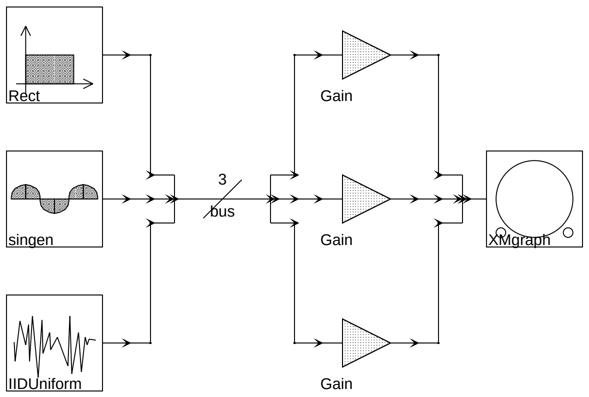


FIGURE 6-14: The Nop star is used to create busses from individual connections, to break busses down into individual lines, and to break out multiportholes into individual ports.

deinterleaves a bus, producing two output busses of equal width. The input bus must have even width. The even numbered input signals are connected to the first output bus, while the odd numbered input signals are connected to the second output bus.

`Nop` Bridge inputs to outputs and then self-destruct. This star is used to split a bus into individual lines or combine individual lines into a bus. It is also used to break out multi-inputs and multi-outputs into individual ports. These icons are labeled “`BusCreate`” and “`BusSplit`”, suggesting their usual function.

If you look inside the icon labeled “Nop” to the right of the above stars, you will open another palette with more icons for the `Nop` stars, shown in figure 6-16.

6.3.2 Map-like stars

`Map` (Two icons.) Map one or more instances of the named block to the input stream(s) to produce the output stream(s). This is implemented by replacing the `Map` star with one or more instances of the named block at preinitialization time. The replacement block(s) are connected as specified by *input_map*

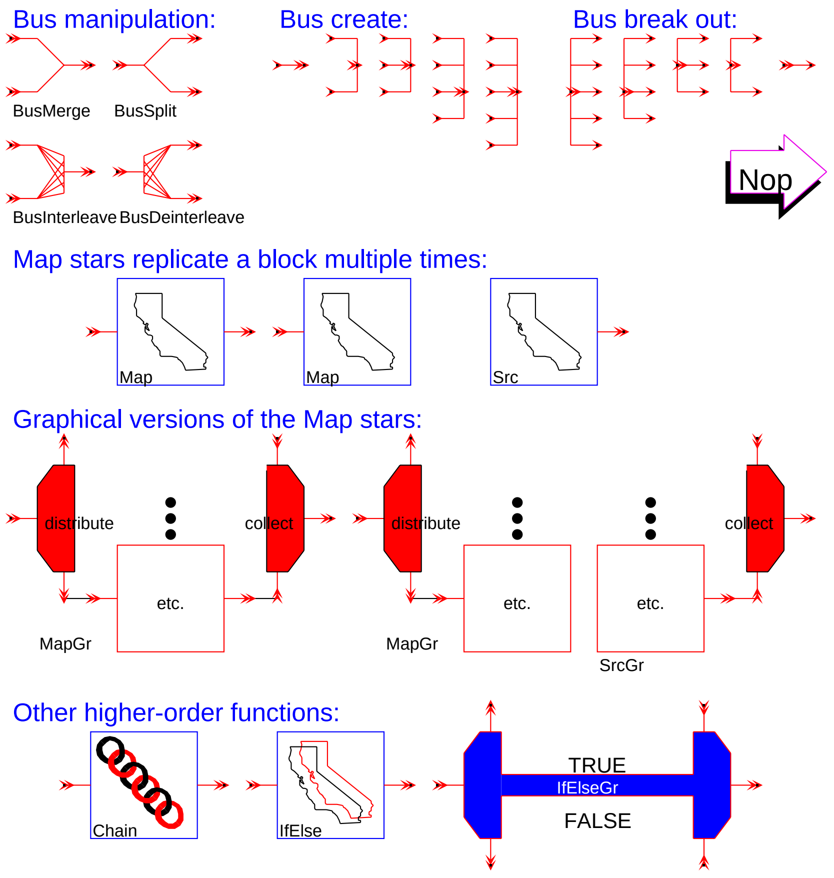


FIGURE 6-15: The top-level palette for the higher-order function stars. The icon labeled “Nop” points to more variants of bus create and bus break ou

	and *output_map*, using the existing connections to the `Map` star. Their parameters are determined by *parameter_map*. See “Setting parameter values” on page 6-6 for examples of the use of *parameter_map*.
`Src`	This is identical to the `Map` star, except that the replacement block is a source block (it has no inputs).
`MapGr`	A variant of the `Map` star where the replacement block is specified by graphically connecting it. There must be exactly one block connected in the position of the replacement block. The `Nop` stars are the only exception: they may be used in addition to the one replacement block in order to control the order of connection.
`SrcGr`	This is identical to the `MapGr` star, except that the replacement block is a source block (it has no inputs)
`Chain`	Create one or more instances of the named block connected in a chain. This is implemented by replacing the `Chain` star with instances of the named blocks at preinitialization time. The replacement block(s) are connected as specified by *input_map*, *internal_map*, and *output_map*. Their parameters are determined by *parameter_map*. If *pipeline* is `YES`, then a unit delay is put on all internal connections.
`IfElse`	This star is just like `Map`, except that it chooses one of two named blocks to replace itself. If the *condition* parameter is `TRUE`, then the *true_block* is used. Otherwise, the *false_block* is used. This can be used to parameterize the use of a given block, or, more interestingly, for statically evaluated recursion.
`IfElseGr`	A variant of the `IfElse` star where the two possible replace-

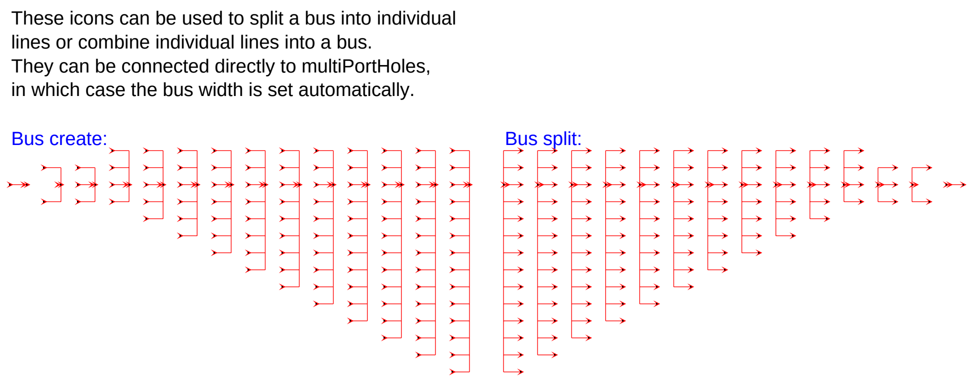

These icons can be used to split a bus into individual lines or combine individual lines into a bus. They can be connected directly to multiPortHoles, in which case the bus width is set automatically.

FIGURE 6-16: A secondary palette with a more complete set of icons for the Nop star. This palette is accessed by looking inside the icon labeled “Nop” in figure 6-15.

ment blocks are specified graphically rather than textually. There must be exactly one block connected in the position of each of the two the replacement blocks. The `Nop` stars are the only exception: they may be used in addition to the two replacement blocks in order to control the order of connection. As of this writing, this star cannot be used with recursion, because pigi will attempt to compile the sub-galaxy before it can be deleted from the schematic by `IfElseGr`.

6.4 An overview of HOF demos

The HOF demos are divided by domain, and are accessed through the demo palette of the individual domain. As of this writing, only the SDF, DDF, DE, and CGC domains have HOF demo palettes.

6.4.1 HOF demos in the SDF domain

The top-level demo palette for the HOF/SDF demos is shown in figure 6-17. The icon labeled "test" points to a set of demos that are not documented here and are used as part of the regression tests in Ptolemy.

`addingSinWaves`

This demo generates a number of sine waves given by the parameter *number_of_sine_waves* and adds them all together. The amplitude of each sine wave is controlled by a Tk slider that is inserted into the control panel when the system is run. The frequency in radians of each sine wave (relative to a sample rate of 2π) is *instance_number* multiplied by π/32. Thus, the first sine wave will have a period of 64 samples. The second will have a period of 32. The third will have a period of 16, etc. The sum of these sine waves is displayed in bar-graph form.

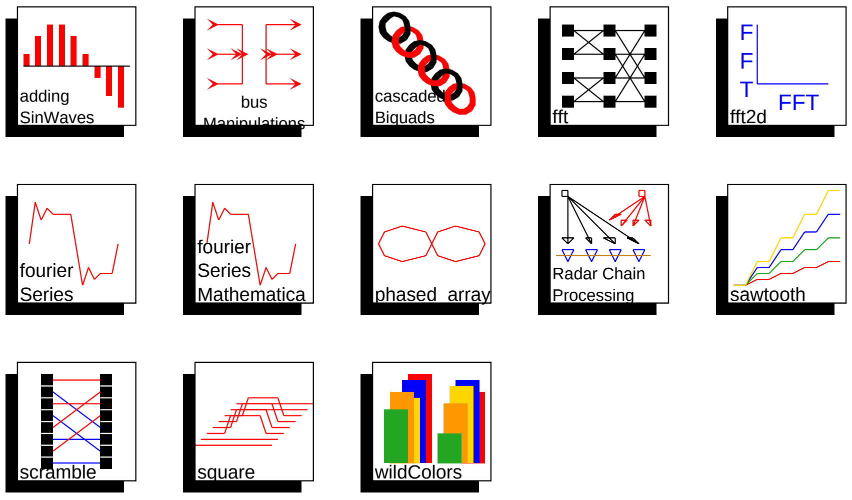


FIGURE 6-17: The top level palette of the HOF demos in the SDF domain.

`busManipulations`
This demo is shown above in figure 6-14 and explained in the accompanying text.

`cascadedBiquads`
The `Chain` HOF star is used to construct a cascade of two second-order direct-form recursive filters (biquads). The frequency response of the cascade is compared against the frequency response of a direct-form fourth-order filter with the same transfer function.

`fft` This system implements a recursive definition of a decimation-in-time fast Fourier transform, comparing its output against that of a direct implementation in C++. The system is configured to use 32 point FFTs to implement a 256 point FFT. The granularity is controllable with the parameters, and can be taken all the way down to the level of multipliers and adders. This system is discussed in detail in [Lee94].

`fft2d` This system generates the same square as in the `square` demo, and then computes its two-dimensional FFT using the method given in figure 6-10.

`fourierSeries` This system generates a number of sinusoids as given by the *number_of_terms* parameter. These are then weighted by the appropriate Fourier series coefficients so that the sum of the sinusoids gives the finite Fourier series approximation for a square wave with period given by the *period* parameter.

`fourierSeriesMma`
This system is similar to the `fourierSeries` system above, but uses Mathematica to calculate parameter values. Mathematica must be licensed on the local workstation for this demo to run.

`phased_array` This system models a planar array of sensors with beamforming and steering, such as might be used with a microphone array or a radar system. The sensors can be positioned arbitrarily in a plane. With the default parameters, 16 sensors are uniformly spaced along the vertical axis, half a wavelength apart, except for one, the fourth, which is offset along the horizontal axis by one tenth of a wavelength. The gain of the array as a function of direction is plotted in both polar and rectangular form (the latter in dB). A Hamming window is applied to the sensor data, as is a steering vector which directs the beam downwards. Zoom into the center of the polar plot to see the effect of the offset sensor. Try changing the *parameter_map* of the left-most `MapGr` higher-order function to realign the offset sensor, and observe the effect on the gain pattern.

`RadarChainProcessing`
This system simulates radar without beamforming. In this simulation, we simulate the effect of an electromagnetic signal traveling from a transmitter to targets and going back to receivers.The delay of the returned signal is used to provide information on the range of the target. The frequency shift, or Doppler effect, is used to provide information on the speed of the target.

	Thus, with these parameters, we estimate the target's properties as in a narrow band radar. The system has been converted from a data parallel form that uses a five-dimensional data array to a functional parallel form that uses higher-order functions to produce streams of streams. The five dimensions are range bin, doppler filters, number of sensors, number of targets and number of pulses. For more information, see `http://ptolemy.eecs.berkeley.edu/papers/Radarsimu.ps.Z`.
`sawtooth`	This demo is shown above in figure 6-6 and explained in the accompanying text.
`scramble`	This system demonstrates the *bit_reverse* galaxy shown above in figure 6-11 and explained in the accompanying text.
`square`	This system demonstrates the `BusMerge` HOF star. It generates an image consisting of a light square on a dark background. The image is first represented using a bus, where each connection in the bus represents one row. The `Commutator` star then rasterizes the image.
`wildColors`	This demo is shown above in figure 6-13 and explained in the accompanying text.

6.4.2 HOF demos in the DE domain

At this time, there are only two simple demos in the DE domain.

`poisson`	This system generates any number of Poisson processes (default 10) and displays them together. To distinguish them, each process produces events with a distinct value.
`exponential`	Combine a number of Poisson processes and show that the interarrival times are exponentially distributed by plotting a histogram. Notice that the histogram bin centered at zero is actually only half as a wide as the others (since the interarrival time cannot be negative), so the histogram displays a value for the zero bin that is half as high as what would be expected.

6.4.3 HOF demos in the CGC domain

The top-level demo palette for the HOF demos in the C Code generation domain (CGC) is shown in figure 6-18.

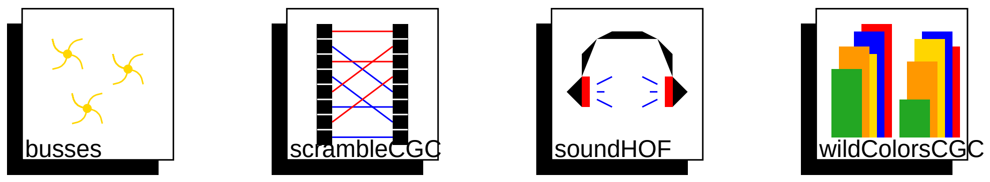


FIGURE 6-18: The top level palette of the HOF demos in the CGC domain.

`busses`	Create a set of ramps of different slopes and display them in both a bar

	chart and using pxgraph.
`scrambledCGC`	This system demonstrates recursion in code generation by taking a ramp in and reordering samples in bit-reversed order.
`soundHOF`	This system produces a sound made by adding a fundamental and its harmonics in amounts controlled by sliders. This demo will work only on Sun workstations.
`wildColorsCGC`	This system is a CGC version of the SDF demo `wildColors`. It creates a number of random sequences and plots them in a pair of bar graphs.

Chapter 7. DDF Domain

Authors: *Soonhoi Ha*
Edward A. Lee
Thomas M. Parks

Other Contributors: *Joseph T. Buck*

7.1 Introduction

The dynamic dataflow (DDF) domain in Ptolemy is a superset of the synchronous dataflow (SDF) and Boolean dataflow (BDF) domains. In the SDF domain, a star consumes and produces a fixed number of particles per invocation (or “firing”). This static information (the number of particles produced or consumed for each star) makes possible compile-time scheduling. In the BDF domain, some actors with data-dependent production or consumption are allowed. The BDF schedulers attempt to construct a compile-time schedule; however, they may fail to do so and fall back on a DDF scheduler. In the DDF domain, the schedulers make no attempt to construct a compile-time schedule. For this reason, there are few constraints on the production and consumption behavior of stars in this domain.

In DDF, a run-time scheduler detects which stars are runnable and fires them one by one until no star is runnable (the system is deadlocked), or until a specified stopping condition has been reached. A star is runnable if it has enough data on its inputs to satisfy its requirements. Thus, the only constraint on DDF stars is that they must specify on each firing how much data they require on each input to be fired again later.

In practice, stars in the DDF domain are written in a slightly simpler way. They are either SDF stars, in which case the number of particles required at each input is a constant, or they are dynamic, in which case they always alert the scheduler before finishing a firing that to be refired they expect some specific number of particles on one particular input. The input that a star is waiting for data on is called the *waitPort*.

Since the DDF domain is a superset of the SDF domain, all SDF stars can be used in the DDF domain. Similarly for BDF stars. Besides the SDF stars, the DDF domain has some DDF-specific stars that will be described in this chapter. The DDF-specific stars overcome the main modeling limitation of the SDF domain in that they can model dynamic constructs such as *conditionals*, *data-dependent iteration,* and *recursion*. All of these except recursion are also supported by the BDF domain. It is even possible, in principle, to dynamically modify a DDF graph as it executes (the implementation of recursion does exactly this). The lower run-time efficiency of dynamic scheduling is the cost that we have to pay for the enhanced modeling power.

Run-time scheduling is expensive. In figure 7-1 we have plotted the execution time of a simple example (setup and run, not including the pigi “compile” or the wrapup). The example contains 17 stars, all simple, all homogeneous synchronous dataflow (producing or consuming a single sample at each port). The tests were run on a Sparc 10 using the `ptrim` executable (on August 26, 1995). The default schedulers in the SDF and DDF domains were used. Note that both schedulers took approximately 13ms at startup, and then exhibited a close to linear increase in execution time. For the SDF scheduler, the slope is approximately 650 μs per iteration, while for the DDF scheduler, it is approximately 1,370 μs per iteration. With 17 stars, this comes to about 38 μs per firing for SDF and 81 μs per firing for DDF. For multirate systems, both of these schedulers will perform poorly compared to the loop scheduler in SDF. Note that for this simple system, DDF is more than twice as expensive as SDF. For systems that require DDF, Ptolemy allows us to regain much of this efficiency by grouping SDF stars in a wormhole that contains an SDF domain. For critical systems that are executed for many iterations, this can provide for considerably faster execution.

There are some subtleties, however, in DDF scheduling. Due to these subtleties, there have been three DDF schedulers implemented, all accessible by setting appropriate target parameters. In the next section, we explain these schedulers.

7.2 The DDF Schedulers

In Ptolemy, a scheduler determines the order of execution of blocks. This would seem to be a simple task in the DDF domain, since there is nothing to do at setup time, and at run time, the scheduler only needs to determine which blocks are runnable and then fire those blocks. Experience dictates, however, that this simple-minded policy is not adequate. In particular, it may use more memory than is required (it may even require an unbounded amount of memory when a bounded amount of memory would suffice). It may also be difficult for a user to specify for how long an execution should proceed.

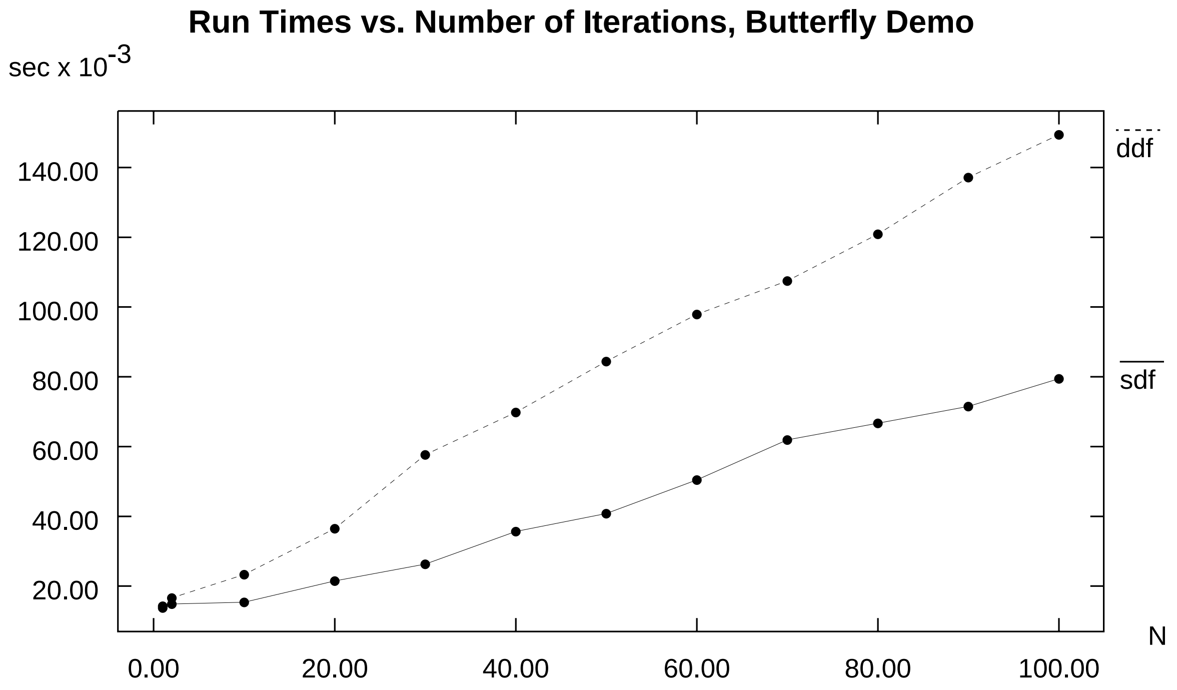


FIGURE 7-1: Time (in milliseconds) vs. number of iterations for the default SDF and default DDF schedulers for a 17 star, single-sample-rate example.

In the SDF domain, an *iteration* is well-defined. It is the minimum number of firings that brings the buffers back to their original state. In SDF, this can be found by a compile-time scheduler by solving the balance equations. In both BDF and DDF, it turns out that it is *undecidable* whether such a sequence of firings exists. This means that no algorithm can answer the question for all graphs of a given size in finite time. This explains, in part, why the BDF domain may fail to construct a compile-time schedule and fall back on the DDF schedulers.

We have three simple and obvious criteria that a DDF scheduler should satisfy:

a. The scheduler should be able to execute a graph forever if it is possible to execute a graph forever. In particular, it should not stop prematurely if there are runnable stars.

b. The scheduler should be able to execute a graph forever in bounded memory if it is possible to execute the graph forever in bounded memory.

c. The scheduler should execute the graph in a sequence of well-defined and determinate iterations so that the user can control the length of an execution by specifying the number of iterations to execute.

Somewhat surprisingly, it turns out to be extremely difficult to satisfy all three criteria at once. The first few versions of the DDF scheduler (up to and including release 0.5.2) did not satisfy (b) or (c). The older scheduler is still available (set the *useFastScheduler* target parameter to *YES*), but its use is not recommended. Its behavior is somewhat unpredictable and sometimes counterintuitive. For example, told to run a graph for one iteration, it may in fact run it forever. Nonetheless, it is still available because it is significantly faster than the newer schedulers. We have not found a way (yet) to combine its efficient and clever algorithm with the criteria above.

The reason that these criteria are hard to satisfy is fundamental. We have already pointed out that it is undecidable whether a sequence of firings exists that will return the graph to its original state. This fact can be used to show that it is undecidable whether a graph can be executed in bounded memory. Thus, no finite analysis can always guarantee (b). The trick is that the DDF scheduler in fact has infinite time to run an infinite execution, so, remarkably, it is still possible to guarantee condition (b). The new DDF schedulers do this.

Regarding condition (a), it is also undecidable whether a graph can be executed forever. This question is equivalent to the *halting problem*, and the DDF model of computation is sufficiently rich that the halting problem cannot always be solved in finite time. Again, we are fortunate that the scheduler has infinite time to carry out an infinite execution. This is really what we mean by dynamic scheduling!

Condition (c) is more subtle and centers around the desire for *determinate* execution. What we mean by this, intuitively, is that a user should be able to tell immediately what stars will fire in one iteration, knowing the state of the graph. In other words, which stars fire should not depend on arbitrary decisions made by the scheduler, like the order in which it examines the stars.

To illustrate that this is a major issue, suppose we naively define an iteration to consist of “firing all enabled stars at most once.” Consider the simple example in figure 7-2. Star A is enabled, so we can fire it. Suppose this makes star B enabled. Should it be fired in the same iteration? Will the order in which we fire enabled stars or determine whether stars are enabled impact the outcome?

We have implemented two policies in DDF. These are explained below.

7.2.1 The default scheduler

The default scheduler, realized in the class `DDFSimpleSched`, first scans all stars and determines which are enabled. In a second pass, it then fires the enabled stars. Thus, the order in which the stars fire has no impact on which ones fire in a given iteration.

Unfortunately, as stated, this simple policy still does not work. Suppose that star A in figure 7-2 produces two particles each time it fires, and actor B consumes 1. Then our policy will be to fire actor A in the first iteration and both A and B in all subsequent iterations. This violates criterion (b), because it will not execute in bounded memory. More importantly, it is counterintuitive. Thus, the `DDFSimpleSched` class implements a more elaborate algorithm.

One iteration, by default, consists of firing all enabled and non-deferrable stars once. If no stars fire, then one deferrable star is carefully chosen to be fired. A *deferrable star* is one with any output arc (except a self-loop) that has enough data to satisfy the destination actor. In other words providing more data on that output arc will not help the downstream actor become enabled; it either already has enough data, or it is waiting for data on another arc. If a deferrable star is fired, it will be the one that has the smallest maximum output buffer sizes. The algorithm is formally given in figure 7-3.

This default iteration is defined to fire actors at most once. Sometimes, a user needs several such *basic iterations* to be treated as a single iteration. For example, a user may wish for a *user iteration* to include one firing of an `XMgraph` star, so that each iteration results in

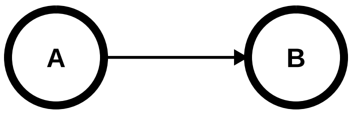


FIGURE 7-2: A simple example used to illustrate the notion of an iteration.

```
At the start of the iteration compute {
      E = enabled actors
      D = deferrable actors
}

One default iteration consists of {
      if (E-D != 0) fire stars in (E-D)
      else if (D != 0) fire the minimax star in D
      else deadlocked.
}
The minimax star is the one with the smallest
maximum number of tokens on its output paths.
```

FIGURE 7-3: The algorithm implementing one basic iteration in the `DDFSimpleSched` class.

one point plotted. The basic iteration may not include one such firing. Another more critical example is a wormhole that contains a DDF system but will be embedded in an SDF system. In this case, it is necessary to ensure that one user iteration consists of enough firings to produce the expected number of output particles.

This larger notion of an iteration can be specified using the target *pragma* mechanism to identify particular stars that must fire some specific number of times (greater than or equal to one) in each user iteration. To use this, make sure the domain is DDF and the target is `DDF-default`. Then in pigi, place the mouse over the icon of the star in question, and issue the *edit-pragmas* command (“a”). One pragma (the one understood by this target) will appear; it is called *firingsPerIteration*. Set it to the desired value. This will then define what makes up an iteration.

7.2.2 The clustering scheduler

If you set the target parameter *restructure* to YES, you will get a scheduler that clusters SDF actors when possible and invokes the SDF scheduler on them. The scheduler is implemented in the class `DDFClustSched`. **WARNING**: As of this writing, this scheduler will not work with wormholes, and will issue a warning. Nonetheless, it is an interesting scheduler for two reasons, the first of which is its clustering behavior. The second is that it uses a different definition of a basic iteration. In this definition, a basic iteration (loosely) consists of as many firings as possible subject to the constraint that no actor fires more than once and that deferrable actors are avoided if possible. The complete algorithm is given in figure 7-4. Use of this scheduler is not advised at this time, however. For one thing, the implementation of clustering adds enough overhead that this scheduler is invariably slower than the default scheduler.

```
The following sets are updated every time a star fires:
      E = enabled actors
      D = deferrable actors
      S = source actors
      F = actors that have fired once already in this iteration

One default iteration consists of:
      while (E-D-F != 0) {
            fire actors in (E-D-F)
      }
      if (F == 0) {
            // All enabled actors are deferrable.
            // Try the non-sources first.
            if (E-S != 0) {
                  fire (E-S);
            } else {
                  fire (S);
            }
      }
      if (F == 0) deadlock
```

FIGURE 7-4: A basic iteration of the DDFClustSched scheduler.

7.2.3 The fast scheduler

In case the new definition of an iteration is inconvenient for legacy systems, we preserve an older and faster scheduler that is not guaranteed to satisfy criteria (b) and (c) above. The basic operation of the fast scheduler is to repeatedly scan the list of stars in the domain and execute the runnable stars until no more stars are runnable, with certain constraints imposed on the execution of sources. For the purpose of determining whether a star is runnable, the stars are divided into three groups. The first group of the stars have input ports that consume a fixed number of particles. All SDF stars, except those with no input ports, are included in this group. For this group, the scheduler simply checks all inputs to determine whether the star is runnable.

The second group consists of the DDF-specific stars where the number of particles required on the input ports is unspecified. An example is the `EndCase` star (a multi-input version of the BDF `Select` star). The `EndCase` star has one control input and one multiport input for data. The control input value specifies which data input port requires a particle. Stars in this group must specify at run time how many input particles they require on each input port. Stars specify a port with a call to a method called *waitPort* and the number of particles needed with a call to *waitNum*. To determine whether a star is runnable, the scheduler checks whether a specified input port has the specified number of particles.

For example, in the `EndCase` star, the *waitPort* points to the *control* input port at the beginning. If the *control* input has enough data (one particle), the star is fired. When it is fired, it checks the value of the particle in the *control* port, and changes the *waitPort* pointer to the input port on which it needs the next particle. The star will be fired again when it has enough data on the input port pointed by *waitPort*. This time, it collects the input particle and sends it to the output port. See Figure 7-5.

The third group of stars comprises sources. Sources are always runnable. Source stars introduce a significant complication into the DDF domain. In particular, since they are always runnable, it is difficult to ensure that they are not invoked too often. This scheduler has a reasonable but not foolproof policy for dealing with this. Recall that the DDF domain is a superset of the SDF domain. The definition of one iteration for this scheduler tries to obtain the same results as the SDF scheduler when only SDF stars are used. In the SDF domain, the number of firings of each source star, relative to other stars, is determined by solving the balance equations. However, in the DDF domain, the balance equations do not apply in the same

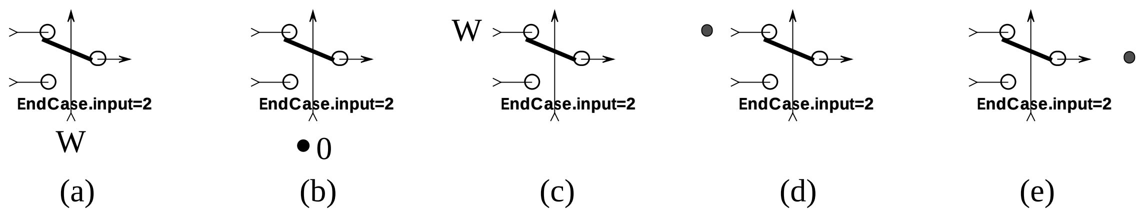


FIGURE 7-5: (a) The EndCase star waits on the *control* port. (b) The star fires when data arrives on the *control* port (the value of the data is 0). (c) Now the star waits for input to arrive on input port 0. (d) The star fires again when data arrives on input port 0. (e) The data that arrived on input port 0 is transmitted by the output port of the EndCase star.

form[1]. The technique we use instead is *lazy-evaluation*.

Lazy evaluation

At the beginning of each iteration of a DDF application, we fire all source stars exactly once, and temporarily declare them "not runnable." We also fire all stars that have enough *initial* tokens on their inputs. After that, the scheduler starts scanning the list of stars in the domain. If a star has some particles on some input arcs, but is not runnable yet, then the star initiates the (lazy) evaluation of those stars that are connected to the input ports requiring more data. This evaluation is "lazy" because it occurs only if the data it produces are actually needed. The lazy-evaluation technique ensures that the relative number of firings of source stars is the same under the DDF scheduler as it would be under the SDF scheduler.

We can now define what is meant by *one iteration* in DDF. An iteration consists of one firing of each source star, followed by as many lazy-evaluation passes as possible, until the system deadlocks. One way to view this (loosely) is that enough stars are fired to consume all of the data produced in the first pass, where the source stars were each fired once. This may involve repeatedly firing some of the source stars. However, a lazy-evaluation is only initiated if a star in need of inputs already has at least one input with enough tokens to fire. Because of this, in some circumstances, the firings that make up an iteration may not be exactly what is expected. In particular, when there is more than one sink star in the system, and the sink stars fire at different rates, the ones firing at higher rates may not be fired as many times as expected. It is also possible for one iteration to never terminate.

When a DDF wormhole is invoked, it will execute one iteration of the DDF system contained in it. This is a serious problem in many applications, since the user may need more control over what constitutes one firing of the wormhole.

7.3 Inconsistency in DDF

So far, we have assumed an error-free program. In the SDF domain, compile-time analysis detects errors due to inconsistent rates of production and consumption of tokens because the balance equations cannot be solved. In DDF, however, such inconsistencies are harder to detect. Our strategy is to detect them at run time, an approach that has two disadvantages. First, it is costly, as will be explained shortly. Second, it is not easy to isolate the sources of errors.

We call a dataflow graph *consistent* if on each arc, in the long run, the same number of particles are consumed as produced [Lee91a]. One source of inconsistency is the sample-rate mismatch that is common to the SDF domain. The DDF domain has more subtle error sources, however, due to the dynamic behavior of DDF stars. In an inconsistent graph, an arc may queue an unbounded number of tokens in the long run. To prevent this, we examine the number of tokens on each arc to detect whether the number is greater than a certain limit (the default is 1024). If we find an arc with too many tokens, we consider it an error and halt the execution. We can modify the limit by setting the target parameter named *maxBufferSize*. The two new schedulers will interpret a negative number here to be infinite capacity. An inconsistent system will run until your computer runs out of memory.

1. Note that the BDF domain adapts the balance equations to the dynamic dataflow case.

The value of the *maxBufferSize* parameter will be the maximum allowed buffer size. Since the source of inconsistency is not unique, isolating the source of the error is usually not possible. We can just point out which arc has a large number of tokens. Of course, if the limit is set too high, some errors will take very long to detect. Note however that there exist perfectly correct DDF systems (which are consistent) that nonetheless cannot execute in bounded memory. It is for this reason that the new schedulers support infinite capacity.

7.4 The default-DDF target

The DDF domain only has one target. The parameters of the target are:

maxBufferSize	(`INT`) Default = `1024` The capacity of an arc (in particles). This is used for the run-time detection of inconsistencies, as explained above. If any arc exceeds this capacity, an error is flagged and the simulation halts. A negative number is interpreted as infinite capacity (unless *useFastScheduler* is `YES`). The value of this parameter does not specify how much memory is allocated for the buffers, since the memory is allocated dynamically.
schedulePeriod	(`FLOAT`) Default = `0.0` This defines the amount of time taken by one iteration (defined above) of the DDF schedule. This is used only for interface with timed domains, such as DE. Note that if you want the count given in the debug panel of the run control panel to indicate the number of iterations, you should set this parameter to one.
runUntilDeadlock	(`INT`) Default = `NO` Unless *useFastScheduler* is set, this modifies the definition of a single iteration to invoke all stars as many times as possible, until the system halts. It is risky to use this because the system may not halt. But in wormholes it is sometimes useful.
restructure	(`INT`) Default = `NO` This specifies that the experimental scheduler `DDFClustSched` should be used. This scheduler attempts to form SDF clusters for more efficient execution. Its use is not advised at this time, however, since it does not work properly with wormholes and is slower than the default scheduler.
useFastScheduler	(`INT`) Default = `NO` This specifies that the older and faster DDF scheduler (from version 0.5.2) should be used. It is difficult, however, to control the length of a run with this scheduler.
numOverlapped	(`INT`) Default = `1` For the fast scheduler only, this gives the number of iteration cycles that can be overlapped for execution. When a DDF system starts up, it normally begins by firing each source star once, as explained above. It then goes into a lazy evaluation mode.

	Setting this parameter to an integer *N* larger than one allows the scheduler to begin with *N* firings of the source stars instead of just one. This can make execution more efficient, because stars downstream from the sources will be able to fire multiple times in each pass through the graph. The default value of this parameter is 1.
logFile	(`STRING`) Default = The default is the empty string. If non-empty, this gives the name of a file to be used for recording scheduler information.

7.5 An overview of DDF stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all of the installed domains. For the DDF domain, the star library is small enough that it is contained entirely in one palette, shown in figure 7-6.

`Case`	(Three icons.) Route an input particle to one of the outputs depending on the control particle. The control particle should be between zero and $N-1$, inclusive, where *N* is the number of outputs.
`EndCase`	(Three icons.) Depending on the control particle, consume a particle from one of the data inputs and send it to the output. The control particle should have value between zero and $N-1$, inclusive, where *N* is the number of inputs.
`DownCounter`	Given an integer input with value *N*, produce a sequence of output integers with values $(N-1)$, $(N-2)$, ... 1, 0.
`LastOfN`	Given a control input with integer value *N*, consume *N* particles from the data input and produce only the last of these at the output.

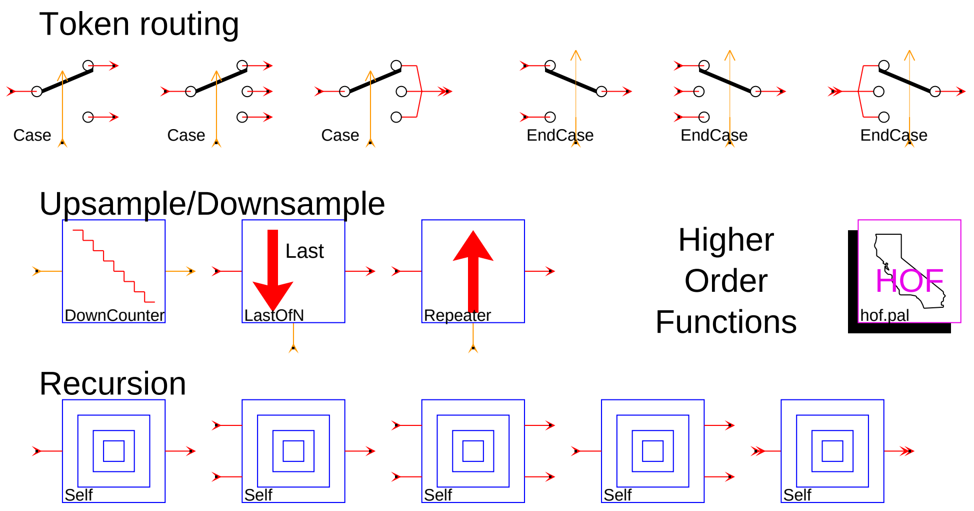


FIGURE 7-6: The palette of stars for the DDF domain.

`Repeater`	Given a control input with integer value *N*, and a single input data particle, produce *N* copies of the data particle on the output.

The Higher Order Functions icon leads to the HOF palette that contains HOF stars that can be used within DDF.

`Self`	(Five icons.) This is a first exploration of recursion and higher-order functions in dataflow. It is still experimental, so do not expect it to be either efficient or bug-free. The star "represents" the galaxy given by the parameter *recurGal*, which must be above it in the hierarchy. That is, when the `Self` star fires, it actually invokes the galaxy that it represents. Since that galaxy is above the `Self` star in the hierarchy, it contains the `Self` star somewhere within it. Thus, this star implements recursion. Since the `Self` star takes an argument (*recurGal*) that specifies the function to invoke, it is itself a higher-order function. The instance of the *recurGal* galaxy is not created until it is actually needed, so the number of instances (the depth of the recursion) does not need to be known *a priori*. If the parameter *reinitialize* is `NO` or `FALSE`, then the instance of the galaxy is created the first time it fires and reused on subsequent firings. If *reinitialize* is `YES` or `TRUE`, then the galaxy is created on every firing and destroyed after the firing. Inputs are sent to the instance of the galaxy and outputs are retrieved from it. The inputs of the named galaxy must be named "input#?" and the outputs must be named "output#?", where "?" is replaced with an integer starting with zero. This allows the inputs and outputs of this star to be matched unambiguously with the inputs and outputs of the referenced galaxy.

7.6 An overview of DDF demos

The demos with icons shown in figure 7-7 illustrate dynamic dataflow principles.

`eratosthenes`	The sieve of Eratosthenes is a recursive algorithm for computing prime numbers. This demo illustrates the implementation of recursion in the DDF domain. This is a concept demonstration only.
`errorDemo`	An example of an inconsistent DDF system. An inconsistent DDF program is one where the long term average number of particles produced on an arc is not the same as the average long term number of particles consumed. This error is detected by bounding the buffer sizes and detecting overflow.
`ifThenElse`	This demo illustrates the use of an SDF wormhole to implement

	a dynamically scheduled construct using the DDF domain. An if-then-else is such a dynamically scheduled construct. The top level schematic represents an SDF system, while the inside schematic represents a DDF system (implementing an if-then-else).
`fibonnacci`	Generate the Fibonnacci sequence using a rather inefficient recursive algorithm that is nonetheless a good example of how to realize recursion.
`loop`	This demo illustrates data-dependent iteration. Input integers are repeatedly multiplied by 0.5 until the product is less than 0.5. Turn on animation to see the iteration.
`picture`	Construct a two-dimensional random walk using a hierarchy of nested wormholes. The outermost SDF domain has a wormhole called “drawline” which internally uses the DDF domain. That wormhole, in turn, has a wormhole called “display” which internally uses the SDF domain.
`repeat`	This simple demo shows the effect of running a DDF scheduler on an SDF system. The *firingsPerIteration* pragma is used to control the meaning of an iteration.
`repeater`	This is a simple illustration of the `Repeater` star, used in an SDF wormhole (DDF inside SDF).
`router`	This is a simple illustration of the `EndCase` star.
`SDFinDDF`	This rather trivial demo illustrates the use of a DDF wormhole whose inside domain is SDF. The top-level system (in the DDF domain) has an if-then-else overall structure, implemented of a matching pair of `Case` and `EndCase` stars. The inside system

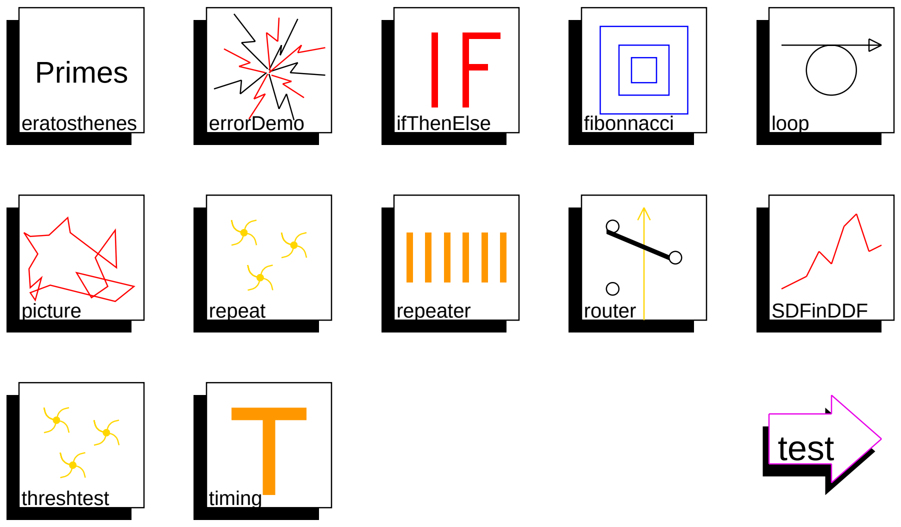


FIGURE 7-7: The DDF demos.

	(in the SDF domain) multiplies the data value by a ramp.
`threshtest`	This demo shows that Karp & Miller style thresholds are supported in DDF. The `Thresh` star is a dummy that implements a settable threshold.
`timing`	This demo illustrates the use of the DDF domain to implement asynchronous signal processing systems. In this case, the system performs baud-rate timing recovery using an approximate minimum mean-square-error (MMSE) technique.

7.7 Mixing DDF with other domains

The mixture of the DDF domain with other domains requires a conversion between different computational models. In particular, some domains associate a *time stamp* with each particle, and other domains do not. Thus, a common function at the `EventHorizon` is the addition of time stamps, the stripping off of time stamps, interpolation between time stamps, or removal of redundant repetitions of identical particles. In this section, DDF-specific features on the domain interface will be discussed.

A galaxy or universe implemented using DDF may have a wormhole which contains a subsystem implemented in another domain. The DDF wormhole looks exactly like a DDF star from the outside. However, there are certain technical restrictions. In particular, it cannot have dynamic input portholes, meaning the number of particles consumed by the wormhole inputs is a compile-time constant. The wormhole is therefore fired when all input ports have new particles. When it is fired, it consumes the input data, invokes the scheduler of the inner domain, and retrieves the output particles. Thus, in all respects except one, the DDF wormhole behaves like an SDF wormhole (see “Wormholes” on page 12-4 for more information). The one exception is that the DDF wormhole need not consistently produce outputs.

When a DDF system is embedded within another domain, you may need to explicitly control what constitutes a firing of the subsystem. Specifically, by setting the *firingsPerIteration* pragma of a star in the DDF subsystem, you control how many firings of that star are required to complete an iteration. Zero means “don’t care.”

Note that some work has been done with a CGDDF target which recognizes and implements certain commonly used programming constructs [Sha92]. See “CG Domain” on page 13-1 for more information.

Chapter 8. BDF Domain

Authors: *Joseph T. Buck*

Other Contributors: *Edward A. Lee*

8.1 Introduction

Boolean-controlled dataflow (BDF) is a domain that can be thought of as a generalization of synchronous dataflow (SDF). It supports dynamic flow of control but still permits much of the scheduling work to be performed at compile time. The dynamic dataflow (DDF) domain, by contrast, makes all scheduling decisions at run time. Thus, while BDF is a generalization of SDF, DDF is still more general. Accordingly, the BDF domain permits SDF actors to be used, and the DDF domain permits BDF actors to be used. This chapter will assume that the reader is familiar with the SDF domain.

The BDF domain can execute any actor that falls into the class of Boolean-controlled dataflow actors. For an actor to be SDF, the number of particles read by each input porthole, or written by each output porthole, must be constant. Under BDF, a generalization is permitted: the number of particles read or written by a porthole may be either a constant or a two-valued function of a particle read on a control porthole for the same star. One of the two values of the function must be zero. The effect of this is that a porthole might read tokens only if the corresponding control particle is zero (`FALSE`) or nonzero (`TRUE`). The control porthole is always of type integer, and it must read or write exactly one particle. Although the particles on the control porthole are of integer type, we treat them as Booleans, using the C/C++ convention that zero is false and nonzero is true.

We say that a porthole that conditionally transfers data based on a control token is a conditional porthole. A conditional input porthole must be controlled by a control input. A conditional output porthole may be controlled by either a control input or a control output. These restrictions permit the run-time flow of control to be determined by looking only at the values of particles on control ports. The compile-time scheduler determines exactly how the flow of control will be altered at run time by the values of these particles. It constructs what we call an *annotated schedule*, which is a compile-time schedule where each firing is annotated with the run-time conditions under which the firing should occur.

The theory that describes graphs of BDF actors and their properties is called the token flow model. Its properties are summarized in [Buc93b] and developed in much more detail in [Buc93c].

The BDF scheduler performs the following functions. First, it performs a consistency check analogous to the one performed by the SDF scheduler to detect certain types of errors corresponding to mismatches in particle flow rates [Lee91a]. Assuming that no error is detected, it then applies a clustering algorithm to the graph, attempting to map it into traditional control structures such as if-then-else and do-while. If this clustering process succeeds in reducing the entire graph to a single cluster, the graph is then executed with the quasi-static schedule corresponding to the clusters. (It is not completely static since some actors will be conditionally executed based on control particle values, but the result is "as static as possible.") If the clustering does not succeed, then the resulting clusters may optionally be executed by the same dynamic scheduler as is used in the DDF domain. Dynamic execution of clusters is enabled or disabled by setting the "allowDynamic" parameter of the default-BDF target.

8.2 The default-BDF target

At this time, there is only one BDF target. The parameters of the target are:

logFile	(`STRING`) Default = The default is the empty string. The filename to which to report various information about a run. If this parameter is empty (the default), there will be no reporting. If the parameter is "<cerr>" or "<cout>", messages will go to the Unix standard error or standard output, respectively.
allowDynamic	(`INT`) Default = `NO` If `TRUE` or `YES`, then dynamic scheduling will be used if the compile-time analysis fails to completely cluster the graph. As shown in [Buc93c], there will always be some valid graphs that cannot be clustered.
requireStronglyConsistent	(`INT`) Default = `NO` If `TRUE` or `YES`, then a graph will be rejected if it is not "strongly consistent" [Lee91a]. This will cause some valid systems, even systems that can be successfully statically scheduled, to be rejected.
schedulePeriod	(`FLOAT`) Default = `10000.0` This defines the amount of time taken by one iteration of the BDF schedule. The notion of "iteration" is defined in the SDF chapter, in the section 5.1.3.

8.3 An overview of BDF stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all of the installed domains. At the current time, the BDF star library is small enough that it is contained entirely in one palette, shown in figure 8-1.

`CondGate`	If the value on the "control" input is nonzero, the input particle is copied to output. Otherwise, no input is consumed (except the control particle) and no output is produced. This is effectively one half of a `Select`.
`Fork`	(Two icons.) Copy the input particle to each output. The SDF fork is not used here because the BDF domain requires some extra steps to assert that each output of a fork is logically equivalent if the input is a Boolean signal.

`Not`	Output the logical inverse of the Boolean input. Again, the equivalent SDF logic block is not adequate because extra steps are needed to assert the logical relationship between the input and the output.
`Select`	If the value on the "control" porthole is nonzero, *N* tokens (from the parameter *N*) from "trueInput" are copied to the output; otherwise, *N* tokens from "falseInput" are copied to the output.
`Switch`	Switch input particles to one of two outputs, depending on the value of the control input. The parameter *N* gives the number of particles read in one firing. If the particle read from the control input is `TRUE`, then the values are written to "trueOutput"; otherwise they are written to "falseOutput".

The Higher Order Functions icon leads to the HOF palette that contains HOF stars that can be used within BDF.

8.4 An overview of BDF demos

The demos with icons shown in figure 8-2 illustrate Boolean-controlled dataflow principles. A useful way to understand these principles when running BDF demos is to display the schedule after a run. This can be done from pigi using the display-schedule command under

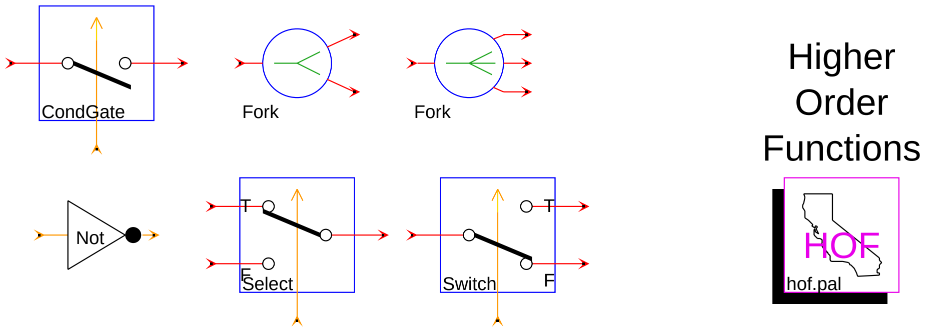


FIGURE 8-1: The palette of stars for the BDF domain. All SDF stars may also be used.

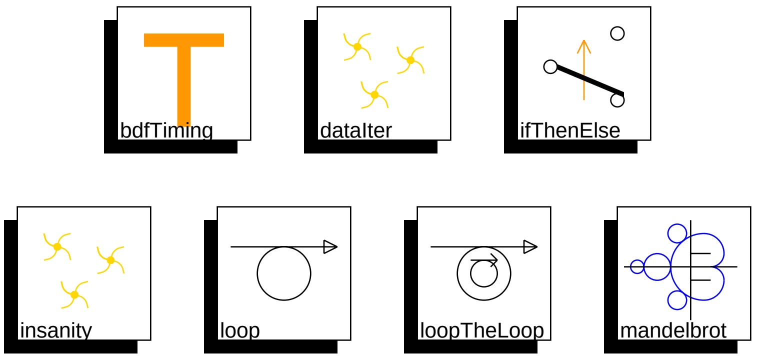


FIGURE 8-2: The BDF demos.

the Exec menu. It must be done before the control panel is dismissed, because dismissing the control panel destroys the scheduler.

`bdfTiming`	This demo is identical to the DDF timing demo, except that it uses BDF `Switch` and `Select` stars instead of DDF `Case` and `EndCase`. The static schedule has some simple if-then constructs to implement conditional firing.
`dataIter`	This simple system, which does nothing interesting, is surprisingly difficult to schedule statically. It requires nesting an if-then within a do-while within a manifest iteration.
`ifThenElse`	This simple system uses `Switch` and `Select` stars to construct an if-then-else.
`insanity`	This peculiar system applies two functions, log and cosine, but the order of application is chosen at random. The BDF clustering algorithm fails to complete on this graph. If the *allowDynamic* parameter of the target is set to `YES`, then the scheduler will construct four SDF subschedules, which must then be invoked dynamically.
`loop`	This system illustrates the classic dataflow mechanism for implementing data-dependent iteration (a do-while). A sequence of integers (a ramp) is the overall input. Each input value gets multiplied by 0.5 inside the loop until its magnitude is smaller than 0.5. Then that smaller result is sent to the output.
`loopTheLoop`	This system is similar to the `loop` demo, except that a second do-while loop is nested within the first.
`mandelbrot`	This system calculates the Mandelbrot set and uses Matlab to plot the output. Matlab must be installed on the local workstation to view the output of this demo, or Matlab must be available on a machine that is accessible via the Unix rsh command. See “Matlab stars” on page 5-26 for more information.

Chapter 9. PN domain

Authors: *Thomas M. Parks*

Other Contributors: *Brian Evans*
Christopher Hylands

9.1 Introduction

In the process network model of computation, concurrent processes communicate through unidirectional first-in first-out channels. This is a natural model for describing signal processing systems where infinite streams of data samples are incrementally transformed by a collection of processes executing in sequence or in parallel. Embedded signal processing systems are typically designed to operate indefinitely with limited resources. Thus, we want to execute process network programs forever with bounded buffering on the communication channels whenever possible. [Par95]

The process network (PN) domain is an experimental implementation of process network model of computation. The PN domain is a superset of the synchronous dataflow (SDF), Boolean dataflow (BDF), and dynamic dataflow (DDF) domains, as shown in Figure 1-2. Thus, any SDF, BDF, or DDF star can be used in the PN domain. In the dataflow subdomains, stars represent dataflow actors, which consume and produce a finite number of particles when they are fired. In the PN domain, stars represent processes, which consume and produce (possibly infinite) streams of particles. When a dataflow actor from the SDF, BDF or DDF domain is used in a PN system, a *dataflow process* is created that repeatedly fires that actor. A separate thread of execution is created for each process. Thread synchronization mechanisms ensure that a thread attempting to read from an empty input is automatically suspended, and threads automatically wake up when data becomes available.

The current implementation of the PN domain is based on a user-level POSIX thread library called Pthreads [Mue93,Mue95]. By choosing the POSIX standard, we improve the portability of the PN domain. Several workstation vendors already include an implementation of POSIX threads in their operating systems, such as Solaris 2.5 and HP-UX 10. Having threads built into the operating system, as opposed to a user-level library implementation, offers the opportunity for automatic parallelization on multiprocessor workstations. That is, the same program would run on uniprocessor workstations and multiprocessor workstations without needing to be recompiled. When multiple processors are available, multiple threads can execute in parallel. Even on uniprocessor workstations, multi-threaded execution offers the advantage that communication can be overlapped with computation.

9.2 Process networks

Kahn describes a model of computation where processes are connected by communication channels to form a network [Kah74,Kah77]. Processes produce data elements or *tokens* and send them along a unidirectional communication channel where they are stored in a first-in first-out order until the destination process consumes them. Communication channels are the *only* method processes may use to exchange information. A set of processes that communicate through a network of first-in first-out queues defines a *program*.

Kahn requires that execution of a process be suspended when it attempts to get data from an empty input channel. A process may not, for example, examine an input to test for the presence or absence of data. At any given point, a process is either *enabled* or it is *blocked* waiting for data on *only one* of its input channels: it cannot wait for data from one channel *or* another. Systems that obey Kahn's model are *determinate*: the history of tokens produced on the communication channels do not depend on the execution order [Kah74]. Therefore, we can apply different scheduling algorithms without affecting the results produced by executing a program.

9.2.1 Dataflow process networks

Dataflow is a model of computation that is a special case of process networks. Instead of using the blocking read semantics of Kahn process networks, dataflow actors have firing rules. These firing rules specify what tokens must be available at the inputs for the actor to fire. When an actor fires, it consumes some finite number of input tokens and produces some finite number of output tokens. For example, when applied to an infinite input stream a firing function *f* may consume just one token and produce one output token:

$$f([x_1, x_2, x_3, \ldots]) = f(x_1)$$

To produce an infinite output stream, the actor must be fired repeatedly. A process formed from repeated firings of a dataflow actor is called a *dataflow process* [Lee95]. The higher-order function *map* converts an actor firing function *f* into a process:

$$map(f)[x_1, x_2, x_3, \ldots] = [f(x)_1, f(x)_2, f(x)_3, \ldots]$$

A higher-order function takes a function as an argument and returns another function. When the function returned by $map(f)$ is applied to the input stream $[x_1, x_2, x_3, \ldots]$ the result is a stream in which the firing function *f* is applied point-wise to each element of the input stream. The *map* function can also be described recursively using the stream-building function *cons,* which inserts an element at the head of a stream:

$$map(f)[x_1, x_2, x_3, \ldots] = cons(f(x_1), map(f)[x_2, x_3, \ldots])$$

The use of map can be generalized so that *f* can consume and produce multiple tokens on multiple streams [Lee95].

Breaking a process down into smaller units of execution, such as dataflow actor firings, makes efficient implementations of process networks possible. The SDF, BDF, and DDF domains implement dataflow process networks by scheduling the firings of dataflow actors. The actor firings of one dataflow process are interleaved with the firings of other processes in a sequence that guarantees the availability of tokens required for each firing. In the PN domain, a dataflow process is created for each dataflow actor. A separate thread of execution is created for each process, and the interleaving of threads is performed automatically. Unlike the dataflow domains, the firing of a dataflow actor is *not* an atomic operation in the PN domain. Because the scheduler does not guarantee the availability of tokens, the firing of an actor can be suspended if it attempts to read data from an empty input channel.

9.2.2 Scheduling dataflow process networks

Because Kahn process networks are determinate, the results produced by executing a program are unaffected by the order in which operations are carried out. In particular, deadlock is a property of the program itself and does not depend on the details of scheduling. Buffer sizes for the communication channels, on the other hand, do depend on the order in which read and write operations are carried out.

For Kahn process networks, no finite-time algorithm can decide whether or not a program will terminate or require bounded buffering. Since we are interested in programs that will never terminate, a scheduler has infinite time to decide these questions. Parks [Par95] developed a scheduling policy that will execute arbitrary Kahn process networks forever with bounded buffering when possible. To enforce bounded buffering, Parks limits channel capacities, which places additional restrictions on the order of read and write operations. Parks reduces the set of possible execution orders to those where the buffer sizes never exceed the capacity limits. In this approach, execution of the entire program comes to a stop each time we encounter artificial deadlock, which can severely limit parallelism. Artificial deadlock occurs when the capacity limits are set too low, causing some processes to block when writing to a full channel. All scheduling decisions are made dynamically during execution.

9.2.3 Iterations in the PN domain

In a complete execution of a program, the program terminates if and only if all processes block attempting to consume data from empty communication channels. Often, it is desirable to have a partial execution of a process network. An iteration in the PN domain is defined such that no actor in a dataflow process will fire more than once. Some actors may not fire, or may fire partially in an iteration if insufficient tokens are available on their inputs.

9.3 Threads

The PN domain creates a separate thread of execution for each node in the program graph. Threads are sometimes called lightweight processes. Modern operating systems, such as Unix, support the simultaneous execution of multiple processes. There need not be any actual parallelism. The operating system can interleave the execution of the processes. Within a single process, there can be multiple lightweight processes or threads, so there are two levels of multi-threading. Threads share a single address space, that of the parent process, allowing them to communicate through simple variables (shared memory). There is no need for more complex, heavyweight inter-process communication mechanisms such as pipes.

Synchronization mechanisms are available to ensure that threads have exclusive access to shared data and cannot interfere with one another to corrupt shared data structures. Monitors and condition variables are available to synchronize the execution of threads. A monitor is an object that can be locked and unlocked. Only one thread may hold the lock on the monitor. If a thread attempts to lock a monitor that is already locked by another thread, then it will be suspended until the monitor is unlocked. At that point, it wakes up and tries again to lock the monitor. Condition variables allow threads to send signals to each other. Condition variables must be used in conjunction with a monitor; a thread must lock the associated monitor before using a condition variable.

9.4 An overview of PN stars

The “open-palette” command in pigi (“O”) will open a checkbox window that you can use to open the standard palettes in all the installed domains. For the PN domain, the star library is small enough that it is easily contained entirely in one palette, shown in figure 9-1

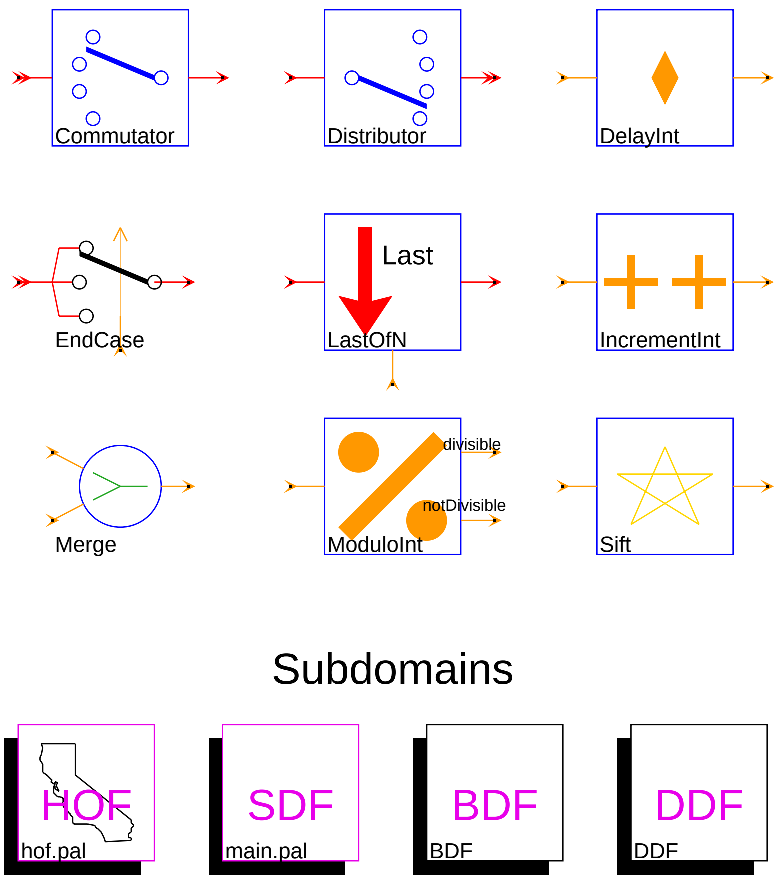


FIGURE 9-1: The palette of stars for the PN domain.

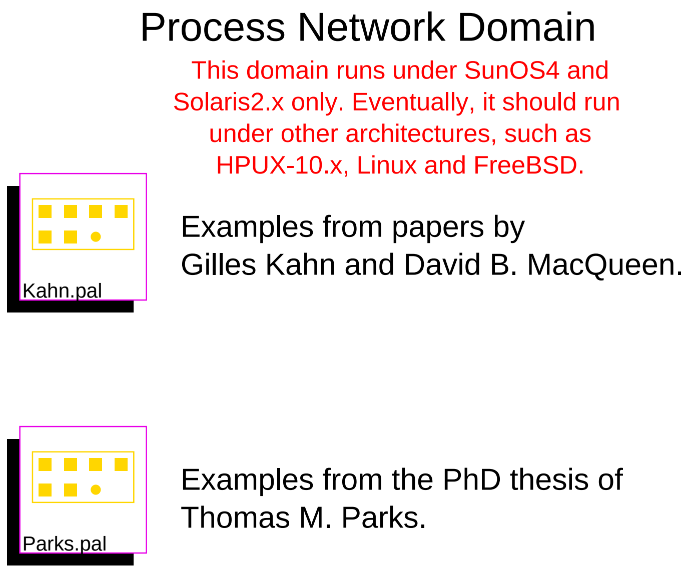
Many of these stars are re-implementations of similarly named stars in the SDF and DDF domains. These implementations take advantage of the multi-threaded nature of execution in the PN domain.

`Commutator`	Takes N input streams (where N is the number of inputs) and synchronously combines them into one output stream. It consumes B particles from an input (where B is the blockSize), and produces B particles on the output, then it continues by reading from the next input. The first B particles on the output come from the first input, the next B particles from the next input, etc.
`Distributor`	Takes one input stream and splits it into N output streams, where N is the number of outputs. It consumes B input particles, where B = blockSize, and sends them to the first output. It consumes another B input particles and sends them to the next

	output, etc.
`DelayInt`	An initializable delay line.
`EndCase`	Depending on the “control” particle, consume a particle from one of the data inputs and send it to the output. The value of the control particle should be between zero and N-1, where N is the number of data inputs.
`LastOfN`	Given a control input with integer value N, consume N particles from the data input and produce only the last of these at the output.
`Merge`	Merge two increasing sequences, eliminating duplicates.
`IncrementInt`	Increment the input by a constant.
`ModuloInt`	Divides the input stream into a stream of numbers divisible by N and another stream of numbers that are not divisible by N.

9.5 An overview of PN demos

There are two subpalettes of PN domain demos, a palette of examples from papers by Gilles Kahn and David B. MacQueen, and a palette of examples from the Ph.D. thesis of Thomas M. Parks. The top-level palette for demos in the Process Network domain is shown in figure 9-2. The subpalettes are described below.



Process Network Domain

This domain runs under SunOS4 and Solaris2.x only. Eventually, it should run under other architectures, such as HPUX-10.x, Linux and FreeBSD.

Examples from papers by Gilles Kahn and David B. MacQueen.

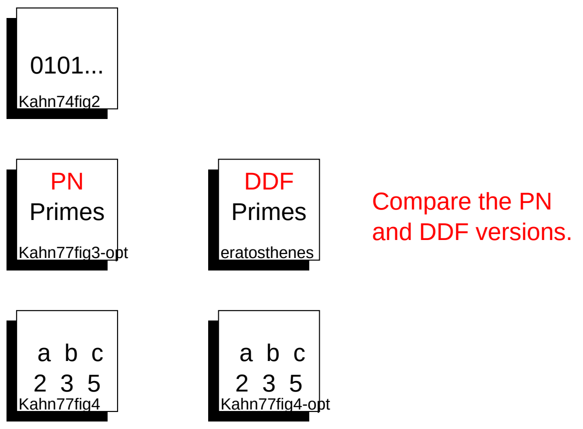
Examples from the PhD thesis of Thomas M. Parks.

FIGURE 9-2: The top-level palette for PN demos.

9.5.1 Examples from papers by Gilles Kahn and David B. MacQueen

These demos are examples from papers by Gilles Kahn and David B. MacQueen. The palette is shown in figure 9-3.

Examples from [Kahn74] and [Kahn77].



[Kahn74] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming", Information Processing, North-Holland Publishing Company, pp 471-475, 1974.

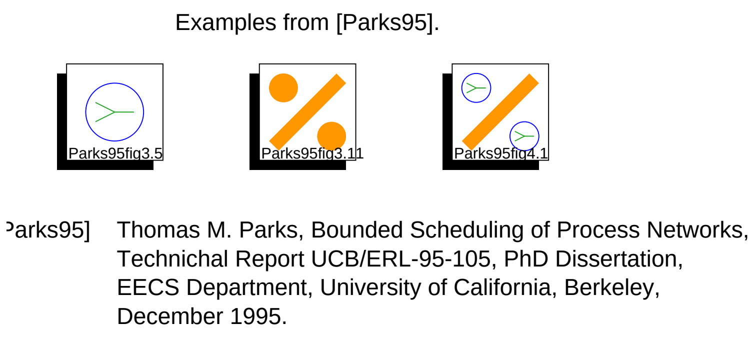
[Kahn77] Gilles Kahn and David B. MacQueen, "Coroutines and Networks of Parallel Processes," Information Processing, North-Holland Publishing Company, pp 993-998, 1977.

FIGURE 9-3: PN domain demos of examples from papers by Gilles Kahn and David B. MacQueen

`Kahn74fig2`	Produce a stream of 0's and 1's. This demo is from figure 2 in [Kah74].
`Kahn77fig3-opt`	Sieve of Eratosthenes with non-recursive sift process. This example shows how process networks can change dynamically during execution. The sift process inserts new filter processes to eliminate multiples of newly discovered primes. This demo is from figure 3 in [Kah77].
`eratosthenes`	Compare this DDF domain demo with the Kahn77fig3-opt demo above.
`Kahn77fig4`	Produce a sequence of integers of the form $2^a 3^b 5^c$. An unbounded number of tokens accumulate in the communication channels as execution progresses. This demo is from figure 4 in [Kah77].
`Kahn77fig4-opt`	Produce a sequence of integers of the form $2^a 3^b 5^c$ with optimizations to avoid generating duplicate values. An unbounded number of tokens accumulate in the communication channels as execution progresses. This demo is from figure 4 in [Kah77].

9.5.2 Examples from the Ph.D. thesis of Thomas M. Parks

These demos are examples from the Ph.D. thesis of Thomas M. Parks. The palette is show in figure 9-4.



Examples from [Parks95].

Parks95] Thomas M. Parks, Bounded Scheduling of Process Networks,
Technichal Report UCB/ERL-95-105, PhD Dissertation,
EECS Department, University of California, Berkeley,
December 1995.

FIGURE 9-4: PN domain demos of examples from the Ph.D. thesis of Thomas M. Parks.

`Parks95fig3.5`	Merge two streams of monotonically increasing integers (multiples of 2 and 3) to produce a stream of monotonically increasing integers with no duplicates. Simple data-driven execution of this example would result in unbounded accumulation of tokens, while demand-driven execution requires that only a small number of tokens be stored on the communication channels. This demo is from figure 3.5 in [Par95].
`Parks95fig3.11`	Separate a stream of monotonically increasing integers into those values that are and are not evenly divisible by 3. Simple demand-driven execution of this example would result in unbounded accumulation of tokens, while data-driven execution requires that only a small number of tokens be stored on the communication channels. This demo is from figure 3.11 in [Par95].
`Parks95fig4.1`	Separate an increasing sequence of integers into those values that are and are not evenly divisible by 5, then merge these two streams to reproduce a stream of increasing integers. Simple data-driven or demand-driven execution of this example would result in unbounded accumulation of tokens. This demo is from figure 4.1 in [Par95].

Chapter 10. SR domain

Authors: *Stephen Edwards*

Other Contributors: *Christopher Hylands*
Mary Stewart

10.1 Introduction

The Synchronous Reactive domain is a statically-scheduled simulation domain in Ptolemy designed for concurrent, control-dominated systems. To allow precise control over timing, it adopts the synchronous model of time, which is logically equivalent to assuming that computation is instantaneous.

10.2 SR concepts

Time in the SR domain is a sequence of instants. In each instant, the system observes its inputs and computes its reaction to them. Each instant is assumed to take no time at all. All computation is treated as being instantaneous.

Communication in the SR domain takes place through unbuffered single driver, multiple receiver channels. In each instant, each channel may have a single event with a value, have no event, or be undefined, corresponding to the case where the system could not decide whether the channel had an event or not. Communication is instantaneous, meaning that if an event is emitted on a channel in a certain instant, every star connected to the channel will see the event in the same instant.

10.3 SR compared to other domains

SR is similar to existing Ptolemy domains, but differs from them in important ways. Like Synchronous Dataflow (SDF), it is statically scheduled and deterministic, but it does not have buffered communication or multi-rate behavior. SR is better for control-dominated systems that need control over when things happen relative to each other; SDF is better for data-dominated systems, especially those with multi-rate behavior.

SR also resembles the Discrete Event (DE) domain. Like DE, its communication channels transmit events, but unlike DE, it is deterministic, statically scheduled, and allows zero-delay feedback loops. DE is better for *modeling* the behavior of systems (i.e., to better understand their behavior), whereas SR is better for *specifying* a system's behavior (i.e., as a way to actually build it).

10.4 The semantics of SR

An SR star must be well-behaved in the following mathematical sense to make SR systems deterministic. It must compute a monotonic function of its inputs, meaning that when it is presented with more-defined inputs, it must produce more-defined outputs. In particular, an output may only switch from undefined to either present or absent when one or more inputs do, but it may not change its value or become undefined.

The semantics of SR are defined as the least fixed point of the system, meaning the least-defined set of values on the communication channels that is consistent with all the stars' functions. That is, if any star were evaluated, it will not want to change its output---the value is already correct. The monotonicity constraint on the stars ensures that there is always exactly one least-defined set, and this is what the SR schedulers calculate.

There are two schedulers for the SR domain, **`default-SR`** and **`dynamic-SR`**. The dynamic scheduler is the easiest to understand. In each instant, it first initializes all the communication channels to "undefined" and then executes all the stars in the system until none of them try to change their outputs. The default scheduler is more shrewd. It uses the communication structure of the system to determine an execution order for the stars that will make them converge. This is based on a topological sort of the stars, but is made more complicated when there are feedback loops.

10.5 Overview of SR stars

The top-level palette is shown in figure 10-1.

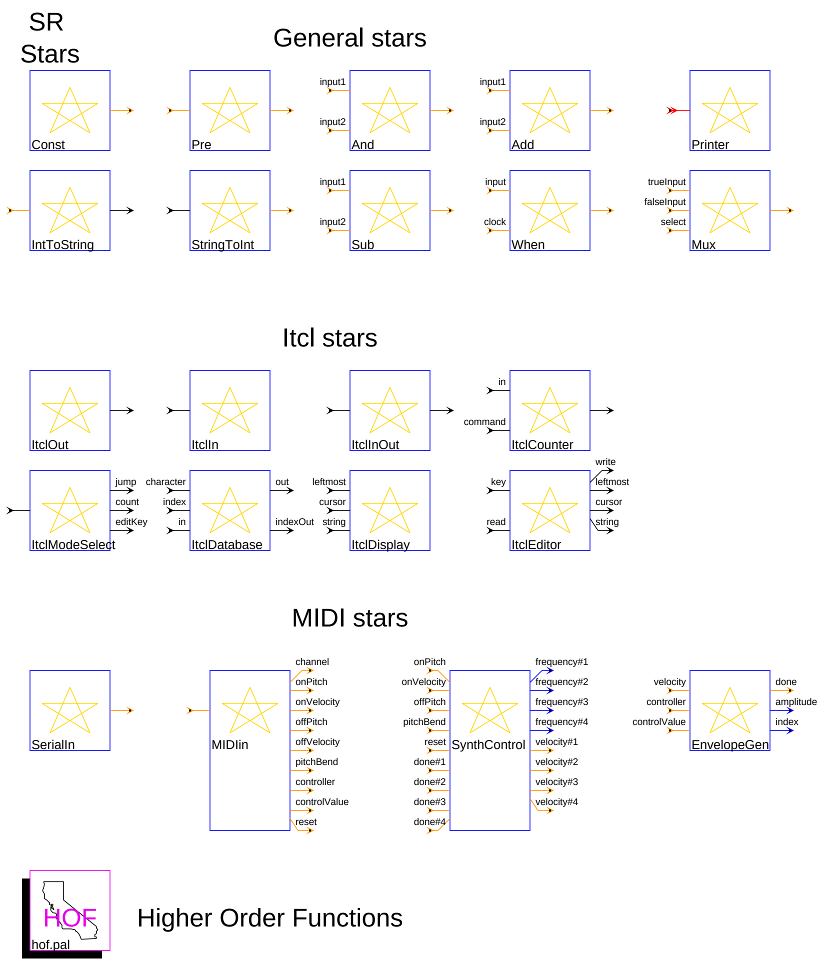


FIGURE 10-1: The top-level palette for SR demos.

10.5.1 General stars

`Const`	Output a constantly-present integer output given by the *level* parameter.
`Pre`	Emit the value of the integer input from the most recent instant in which it was present.
`And`	Emits the logical AND of the two integer inputs, or absent if both inputs are absent.
`Add`	Emit the sum of the two integer inputs, or the value of the present input if the other is absent.

`Printer`	Print the value of each input to the file specified by the *fileName* state. All inputs are printed on a single line with the prefix in the *prefix* state.
`IntToString`	Convert the integer input to a string.
`StringToInt`	Convert the string input to an integer.
`Sub`	Emits the different of the two integer inputs, or absent if both inputs are absent.
`When`	When the clock is true, emit the input on the output, otherwise, leave the clock absent.
`Mux`	Depending on the value of the select input, copy either the true input or the false input to the output.

10.5.2 Itcl stars

By default, all of these stars have no behavior. They provide an interface for user-written Itcl scripts that specify their behavior. Each has the following states: *itclClassName*, the class name of the itcl object associated with the star, *itclSourceFile*, the path name of the itcl script containing the definition of this class, and *itclObjectName*, the name of the itcl object to create. If this field is left blank, the object is given the name of the star instance. See the SR chapter in the Programmer's Manual for more information.

`ItclOut`	Single output Itcl star.
`ItclIn`	Single input Itcl star.
`ItclInOut`	One input, one output Itcl star.
`ItclCounter`	Itcl incrementer/decrementer.
`ItclModeSelect`	Itcl star used in the Rolodex demo.
`ItclDatabase`	Simple sorted database.
`ItclDisplay`	Display for the rolodex.
`ItclEditor`	Editor for the rolodex

10.5.3 MIDI stars

The MIDI stars below are used in the MIDISynthesizer demo described later in this chapter. We include these stars only as an example of what can be done with the Synchronous Reactive domain.

`SerialIn`	Emit the character waiting on the serial port, or leave the output absent if there isn't one. The *deviceName* state specifies the port, and *baudRate* specifies the speed.
`MIDIin`	An interpreter for the MIDI protocol. It takes an incoming MIDI stream and fans it out to Note On and Note Off commands.
`SynthControl`	A polyphonic synthesizer control.

`EnvelopeGen`	An envelope generator for FM sound synthesis.

10.6 An overview of SR demos

There are currently three SR demos. The palette is shown in figure.

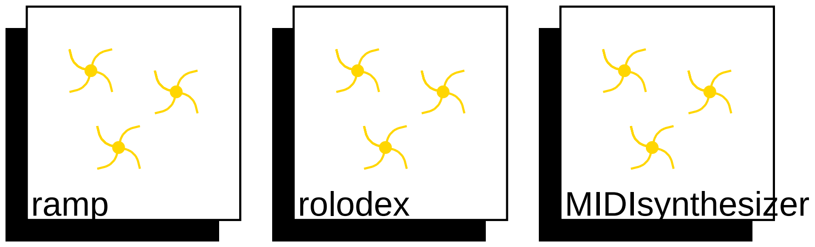


FIGURE 10-2: Synchronous Reactive demos.

`ramp`	Prints a sequence of increasing integers. Essentially a "hello world" for the SR domain, this demonstrates how Pre can interact with an adder.
`rolodex`	A digital address book implemented in SR. This demonstrates how itcl stars can be used to prototype user-interface-dominated systems at a high level. The system is divided into keyboard, database, editor, and display blocks.
`MIDIsynthesizer`	A music synthesizer written in SR. This is a polyphonic sound synthesizer written using custom SR stars for the control portion. Waveform synthesis is done using an FM algorithm implemented in CGC. This requires a MIDI keyboard to be attached to `/dev/ttya`, and functioning CGC audio drivers for your platform. More information on demonstrating the Midi stars with a Yamaha keyboard controller follows.

10.6.1 Use of the Yamaha CBX-K1XG as a midi keyboard controller

Setting Up Hardware Connections:

- The keyboard has an external power supply. Connect one end of the power supply to the DC IN jack on the rear panel of the keyboard and connect the other to a suitable electrical outlet.
- Connect the `TO HOST` Terminal of the keyboard to the serial port of the Sparcstation using an 8-pin Mini-DIN to D-Sub 25-pin cable. You will probably need to use a null modem with this connection.
- Select `PC2` (38,400 baud) with the `Host Select Switch`, located on the rear panel of the midi keyboard.
- You may need to use a Y cable if the Sparcstation to which you are connecting has more than one port (A-B).

*Note that it is necessary to have functioning CGC audio drivers in order to demo the SR stars written for a keyboard controller.

Operation of the Yamaha keyboard controller

A sequence of keys specific to Yamaha’s controller architecture must be triggered to send midi messages to the SR stars to generate synthesized sound within Ptolemy. See the Yamaha user’s manual for detailed instructions

Chapter 11. Finite State Machine Domain

Authors: *Bilung Lee*

Other Contributors: *Christopher Hylands*
Mary Stewart

11.1 Introduction

The finite state machine (FSM) has been one of the most popular models for describing control-oriented systems, e.g., real-time process controllers. A directed node-and-arc graph, called a state transition diagram (STD), can be used to describe an FSM. An STD represents a system in the form of states (nodes) and transitions (arcs) between states.

11.2 Graphical User Interface

The original visual interface to Ptolemy, called Vem, is not suitable for drawing the STD for an FSM. A new visual editor based on Tycho, a hierarchical syntax manager and part of the Ptolemy project, is being developed based on drawing mechanisms created by Wan-teh Chang.

11.2.1 Edit a new STD file

A new STD file can be created from a Tycho editor that is running inside Ptolemy or from a standalone Tycho process. To start Tycho from within Ptolemy, type a `y`. To start a standalone Tycho process, run `tycho` from the shell prompt. Once Tycho is started, either open a file with the file extension `.std`, or select the Tycho `Window` menu and choose `State transition diagram editor`. A message window will pop up to ask for machine type. Currently there is one type of FSM supported, called mixed `Mealy/Moore machine`.

11.2.2 Edit the Input/Output and Internal Events Names

If there are any input/output and internal events for the FSM, their names must be specified as follows: Select the `Special` menu and choose `I/O Port Names...`, then enter the names for the input/output. Each name should be separated by at least one space.

11.2.3 Draw/Edit a State

To draw a state, either select the `Edit` menu and choose `New Node`, or use the keyboard accelerator `N` key. A crossbar cursor will appear in the window. Press and hold (don't release) the left mouse button and move the mouse to get a different shape of node. Release the button to finish the drawing.

To edit a state, first select the state by pressing the left mouse button on your selection. Then either select the `Edit` menu and choose `Edit Item`, or use the keyboard shortcut and type an `e`.

11.2.4 Draw/Edit a Transition

To draw a transition, either select the `Edit` menu and choose `New Arc`, or use the keyboard shortcut and type an `A`. A crossbar cursor will appear in the window. Press the left mouse button on the periphery of the starting state. Move the mouse and press the left mouse button to get a more delicate arc. To finish the drawing, move the mouse on the periphery of the ending state and press the left mouse button.

To edit a transition, select the transition by pressing the left mouse button. Then either select the `Edit` menu and choose `Edit Item`, or use the keyboard shortcut and type an `e`.

11.2.5 Delete a State/Transition

First select the state/transition by pressing the left mouse button on your selection. Then select the `Edit` menu and choose `Delete`.

11.2.6 Move/Reshape a State/Transition

To move a state, press and hold the left mouse button on the state and then move the mouse to the desired position. To reshape a state, first select the state, and then move the cursor to the up/down/left/right periphery of the state to get an up/down/left/right arrow-shape cursor. Press and hold the left mouse button and move the mouse to reshape it.

To move/reshape a transition, first select it, then some small rectangles will appear along the arc. Select and move the rectangles to move/reshape the arc.

11.2.7 Slave Processes of States

In an FSM, each state may be associated with a slave process. This slave process could be a subsystem of any other Ptolemy domain or be another FSM Galaxy. To specify the slave process of a state, when editing a state, give the full path name and file name of the subsystem. For example, a Vem facet could be specified for the Galaxy of any other Ptolemy domain, or an STD file could be specified for the FSM Galaxy.

11.3 Working within Ptolemy

In terms of implementation, a standalone FSM domain in Ptolemy is not very interesting. The reason for this is that most applications, in addition to control, contain many other features, e.g., signal processing. Moreover, there are other Ptolemy domains with which an FSM can interact. By mixing the FSM domain with other domains, we get a very powerful FSM model. Currently the FSM domain only works with the SDF and DE domains.

11.3.1 Make an Icon in Vem

Unlike other domains, an FSM galaxy is edited in a `.std` file using Tycho instead of Vem. However, to work with other domains, an icon in Vem is required to represent the corresponding FSM galaxy described in an STD file. To make an icon in Vem for the STD file, first start Tycho from within Ptolemy by pressing `y`. Open the file in Tycho and choose `Make Icon...`.from the `Special` pull-down menu. This will load the FSM galaxy into the Ptolemy kernel and generate an icon with appropriate input/output ports in the specified palette in Vem.

The icon looks like a star (blue outline) but it is actually a galaxy. This may be confusing, but the idea is to avoid using an Octtools handle. There are two different ways to make an icon in Vem: `ptkSetMkSchemIcon` and `ptkSetMkStar`. The former needs an `OctFacetHandle` as one argument and is used for other “Vem” galaxies. However, since the FSM galaxy uses Tycho as the editor instead of Vem, there is no `OctFacetHandle`. Therefore the latter (`ptksetMkStar`), which uses the star (or galaxy) name instead, is more appropriate.

11.3.2 Look Inside an FSM Galaxy

Similar to the galaxy icon in the other domain, when the `i` key is pressed on the icon to look inside the galaxy, the corresponding STD file which describes the FSM galaxy will be automatically invoked in Tycho. (Note: if the environment variable `PT_DISPLAY` is set to another editor instead of Tycho, it must be unset for the look-inside to work.)

11.3.3 Compile an FSM Galaxy

The FSM domain uses EditSTD to edit a Galaxy, but other domains use Vem as the editor. Therefore, when an FSM Galaxy is compiled, the EditSTD (Tycho) is invoked to compile the state transition graph into the Ptolemy kernel.

11.4 An overview of FSM demonstrations

There are currently three FSM demos illustrated in figure.

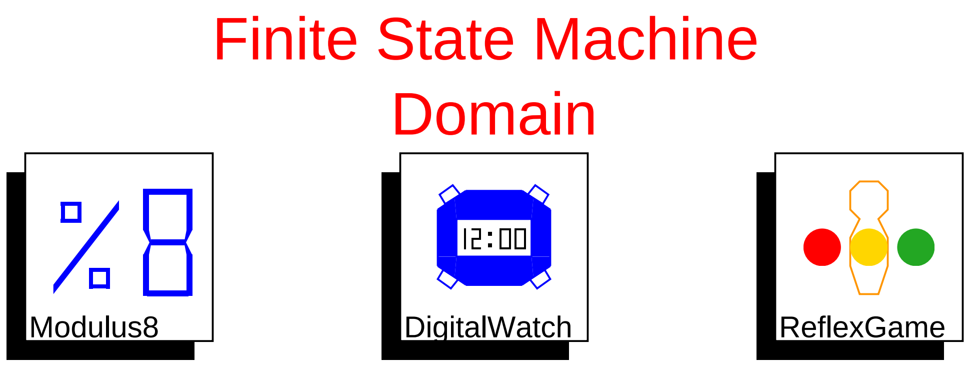


FIGURE 11-1: Finite State Machine demos.

`modulus8`	A three-bit counter with initialization and interruption mechanisms. This demonstrates how the three-bit counter is built by three one-bit counters communicating in the SDF domain.

`reflexGame`	A simple game to test the reflexes of the player. The top-level DE domain models a clock, while the FSM domain models the various states in the reflex game. The SDF domain is used for numerical computations.
`digiWatch`	A digital watch example. This demonstrates that a system with sophisticated control can be achieved by hierarchically nesting the FSM domain.

11.5 Current Limitations

As of Ptolemy release 0.7, this domain is still experimental and is not yet fully developed. Under current implementation, the FSM domain can only be used as a Galaxy. Thus, the FSM domain has to work with other domains that support the inputs to the FSM Galaxy and display the outputs generated by the FSM Galaxy. However, it is possible to implement some blocks that provide the inputs and that display the outputs in the FSM domain.

An FSM system terminates when it reaches a final state, but this feature is not yet implemented. Further, since the FSM system must currently be embedded in other domains under current implementation, the termination of a system is controlled by the topmost domain.

Chapter 12. DE Domain

Authors:	*Joseph T. Buck*
	Soonhoi Ha
	Paul Haskell
	Edward A. Lee
	Thomas M. Parks
Other Contributors:	*Anindo Banerjea*
	Philip Bitar
	Rolando Diesta
	Brian L. Evans
	Richard Han
	Christopher Hylands
	Ed Knightly
	Tom Lane
	Gregory S. Walter

12.1 Introduction

The discrete event (DE) domain in Ptolemy provides a general environment for time-oriented simulations of systems such as queueing networks, communication networks, and high-level models of computer architectures. In this domain, each `Particle` represents an *event* that corresponds to a change of the system state. The DE schedulers process events in chronological order. Since the time interval between events is generally not fixed, each particle has an associated *time stamp*. Time stamps are generated by the block producing the particle based on the time stamps of the input particles and the latency of the block.

12.2 The DE target and its schedulers

The DE domain, at this time, has only one target. This target has three parameters:

timeScale	(`FLOAT`) Default = `1.0` A scaling factor relating local simulated time to the time of other domains that might be communicating with DE.
syncMode	(`INT`) Default = `YES` An experimental optimization explained below, again aimed at mixed-domain systems.
calendar queue scheduler?	(`INT`) Default = `YES` A Boolean indicating whether or not to use the faster “calendar queue” scheduler, explained below.

The DE schedulers in Ptolemy determine the order of execution of the blocks. There are two schedulers that have been implemented which are distributed with the domain. They expect particular behavior (operational semantics) on the part of the stars. In this section, we describe the semantics.

12.2.1 Events and chronology

A DE star models part of a system response to a change in the system state. The change of state, which is called an *event*, is signaled by a particle in the DE domain. Each particle is assigned a time stamp indicating when (in simulated time) it is to be processed. Since events are irregularly spaced in time and system responses are generally very dynamic, all scheduling actions are performed at run-time. At run-time, the DE scheduler processes the events in chronological order until simulated time reaches a global "stop time."

Each scheduler maintains a *global event queue* where particles currently in the system are sorted in accordance with their time stamps; the earliest event in simulated time being at the head of the queue. The difference between the two schedulers is primarily in the management of this event queue. Anindo Banerjea and Ed Knightly wrote the default DE Scheduler, which is based on the "calendar queue" mechanism developed by Randy Brown [Bro88]. (This was based on code written by Hui Zhang.) This mechanism handles large event queues much more efficiently than the alternative, a more direct DE scheduler, which uses a single sorted list with linear searching. The alternative scheduler can be selected by changing a parameter in the default DE target.

Each scheduler fetches the event at the head of the event queue and sends it to the input ports of its destination block. A DE star is executed (fired) whenever there is a new event on any of its input portholes. Before executing the star, the scheduler searches the event queue to find out whether there are any simultaneous events at the other input portholes of the same star, and fetches those events. Thus, for each firing, a star can consume all simultaneous events for its input portholes. After a block is executed it may generate some output events on its output ports. These events are put into the global event queue. Then the scheduler fetches another event and repeats its action until the given stopping condition is met.

It is worth noting that the particle movement is not through `Geodesics`, as in most other domains, but through the global queue in the DE domain. Since the geodesic is a FIFO queue, we cannot implement the incoming events which do not arrive in chronological order if we put the particles into geodesics. Instead, the particles are managed globally in the event queue.

12.2.2 Event generators

Some DE stars are event generators that do not consume any events, and hence cannot be triggered by input events. They are first triggered by system-generated particles that are placed in the event queue before the system is started. Subsequent firings are requested by the star itself, which gives the time at which it wishes to be refired. All such stars are derived from the base class `RepeatStar`.

`RepeatStar` is also used by stars that do have input portholes, but also need to schedule themselves to execute at particular future times whether or not any outside event will arrive then. An example is `PSServer`.

In a `RepeatStar`, a special hidden pair of input and output ports is created and connected together. This allows the star to schedule itself to execute at any desired future time(s), by emitting events with appropriate time stamps on the feedback loop port. The hidden ports are in every way identical to normal ports, except that they are not visible in the graphical user interface. The programmer of a derived star sometimes needs to be aware that these ports are present. For example, the star must not be declared to be a delay star (meaning that no input port can trigger a zero-delay output event) unless the condition also holds for the feedback port (meaning that refire events don't trigger immediate outputs either). See the Programmer's Manual for more information on using `RepeatStar`.

12.2.3 Simultaneous events

A special effort has been made to handle simultaneous events in a rational way. As noted above, all available simultaneous events at all input ports are made available to a star when it is fired. In addition, if two distinct stars can be fired because they both have events at their inputs with identical time stamps, some choice must be made as to which one to fire. A common strategy is to choose one arbitrarily. This scheme has the simplest implementation, but can lead to unexpected and counter-intuitive results from a simulation.

The choice of which to fire is made in Ptolemy by statically assigning priorities to the stars according to a topological sort. Thus, if one of two enabled stars could produce events with zero delay that would affect the other, as shown in figure 12-1, then that star will be fired first. The topological sort is actually even more sophisticated than we have indicated. It follows triggering relationships between input and output portholes selectively, according to assertions made in the star definition. Thus, the priorities are actually assigned to individual portholes, rather than to entire stars. See the Programmer's Manual for further details.

There is a pitfall in managing time stamps. Two time stamps are not considered equal unless they are exactly equal, to the limit of double-precision floating-point arithmetic. If two time stamps were computed by two separate paths, they are likely to differ in the least significant bits, unless all values in the computation can be represented exactly in a binary representation. If simultaneity is critical in a given application, then exact integral values should be used for time stamps. This will work reliably as long as the integers are small enough to be represented exactly as double-precision values. Note that the DE domain does not enforce integer timestamps --- it is up to the stars being used to generate only integer-valued event timestamps, perhaps by rounding off their calculated output event times.

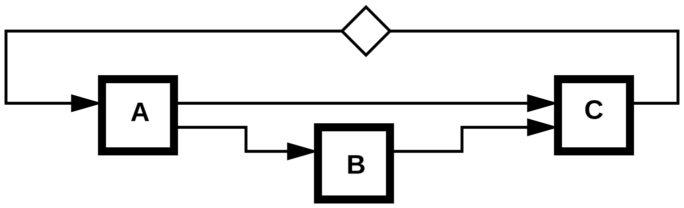


FIGURE 12-1: When DE stars are enabled by simultaneous events, the choice of which to fire is determined by priorities based on a topological sort. Thus if B and C both have events with identical time stamps, B will take priority over C. The delay on the path from C to A serves to break the topological sort.

12.2.4 Delay-free loops

Many stars in the DE domain produce events with the same time stamps as their input events. These zero-delay stars can create some subtleties in a simulation. An *event-path* consists of the physical arcs between output portholes and input portholes plus zero-delay paths inside the stars, through which an input event instantaneously triggers an output event. If an event-path forms a loop, we call it a *delay-free loop*. While a delay-free loop in the SDF domain results in a deadlock of the system, a delay-free loop in the DE domain potentially causes unbounded computation. Therefore, it is advisable to detect the delay-free loop at compile-time. If a delay-free loop is detected, an error is signaled.

Detecting delay-free loops reliably is difficult. Some stars, such as `Server` and `Delay`, take a parameter that specifies the amount of delay. If this is set to zero, it will fool the scheduler. It is the user's responsibility to avoid this pathological case. This is a special case of a more general problem, in which stars conditionally produce zero-delay events. Without requiring the scheduler to know a great deal about such stars, we cannot reliably detect zero-delay loops. What appears to be a delay-free path can be safe under conditions understood by the programmer. In such situations, the programmer can avoid the error message placing a delay element on some arc of the loop. The delay element is the small green diamond found at the top of every star palette in Pigi. *It does not actually produce any time delay in simulated time*. Instead, it declares to the scheduler that the arc with the delay element should be treated as if it had a delay, even though it does not. A delay element on a directed loop thus suppresses the detection of a delay-free loop.

Another way to think about a delay marker in the DE domain is that it tells the scheduler that it's OK for a particle crossing that arc to be processed in the "next" simulated instant, even if the particle is emitted with timestamp equal to current time. Particles with identical timestamps are normally processed in an order that gives dataflow-like behavior within a simulated instant. This is ensured by assigning suitable firing priorities to the stars. A delay marker causes its arc to be ignored while determining the dataflow-based priority of star firing; so a particle crossing that arc triggers a new round of dataflow-like evaluation.

12.2.5 Wormholes

"Time" in the DE domain means simulated time. The DE domain may be used in combination with other domains in Ptolemy, even if the other domains do not have a notion of simulated time. A given simulation, therefore, may involve several schedulers, some of which use a notion of simulated time, and some of which do not. There may also be more than one DE scheduler active in one simulation. The notion of time in the separate schedulers needs to be coordinated. This coordination is specific to the inner and outer domains of the wormhole. Important cases are described below.

SDF within DE

A common combination of domains pairs the SDF domain with the DE domain. There are two possible scenarios. If the SDF domain is inside the DE domain, as shown in figure 12-2, then the SDF subsystem appears to the DE system as a zero-delay block. Suppose, for example, that an event with time stamp T is available at the input to the SDF subsystem. Then when the DE scheduler reaches this time, it fires the SDF subsystem. The SDF subsystem runs the SDF scheduler through one iteration, consuming the input event. In response, it will typically produce one output event, and this output event will be given the time stamp T.

If the SDF subsystem in figure 12-2 is a multirate system, the effects are somewhat more subtle. First, a single event at the input may not be sufficient to cycle through one iteration of the SDF schedule. In this case, the SDF subsystem will simply return, having produced no output events. Only when enough input events have accumulated at the input will any output events be produced. Second, when output events are produced, more than one event may be produced. In the current implementation, all of the output events that are produced have the same time stamp. This may change in future implementations.

More care has to be taken when one wants an SDF subsystem to serve as a source star in a discrete-event domain. Recall that source stars in the DE domain have to schedule themselves. One solution is to create an SDF “source” subsystem that takes an input, and then connect a DE source to the input of the SDF wormhole. We are considering modifying the wormhole interface to support mixing sources from different domains automatically.

DE within SDF

The reverse scenario is where a DE subsystem is included within an SDF system. The key requirement, in this case, is that when the DE subsystem is fired, it must produce output events, since these will be expected by the SDF subsystem. A very simple example is shown in figure 12-3. The DE subsystem in the figure routes input events through a time delay. The events at the output of the time delay, however, will be events in the future. The `Sampler` star,

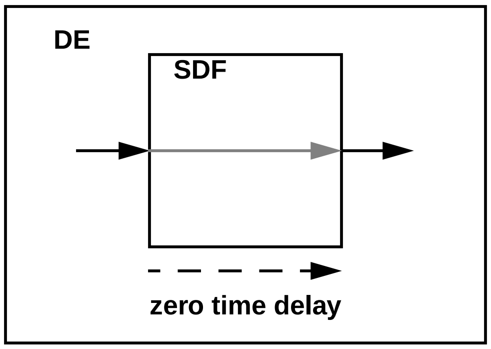


FIGURE 12-2: When an SDF domain appears within a DE domain, events at the input to the SDF subsystem result in zero-delay events at the output of the SDF subsystem. Thus, the time stamps at the output are identical to the time stamps at the inputs.

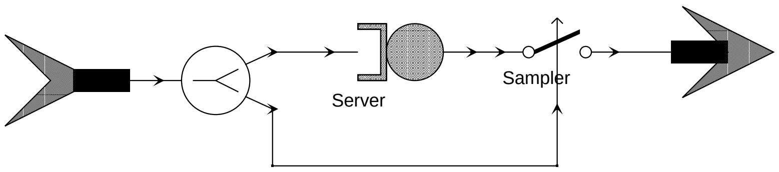


FIGURE 12-3: A typical DE subsystem intended for inclusion within an SDF system. When a DE subsystem appears within an SDF subsystem, the DE subsystem must ensure that the appropriate number of output events are produced in response to input events. This is typically accomplished with a “Sampler” star, as shown.

therefore, is introduced to produce an output event at the current simulation time. This output event, therefore, is produced before the DE scheduler returns control to the output SDF scheduler.

The behavior shown in figure 12-3 may not be the desired behavior. The `Sampler` star, given an event on its control input (the bottom input), copies the most recent event from its data input (the left input) to the output. If there has been no input data event, then a zero-valued event is produced. There are many alternative ways to ensure that an output event is produced. For this reason, the mechanism for ensuring that this output event is produced is not built in. The user must understand the semantics of the interacting domains, and act accordingly.

Timed domains within timed domains

The DE domain is a timed domain. Suppose it contains another timed domain in a DE wormhole. In this case, the inner domain may need to be activated at a given point in simulated time even if there are no new events on its input portholes. Suppose, for instance, that the inner domain contains a clock that internally generates events at regular intervals. Then these events need to be processed at the appropriate time regardless of whether the inner system has any new external stimulus.

The mechanism for handling this situation is simple. When the internal domain is initialized or fired, it can, before returning, place itself on the event queue of the outer domain, much the same way that an event generator star would. This ensures that the inner event will be processed at the appropriate time in the overall chronology. Thus, when a timed domain sits within a timed domain wormhole, before returning control to the scheduler of the outer domain, it requests rescheduling at the time corresponding to the oldest time stamp on its event queue, if there is such an event.

When an internal timed domain is invoked by another time domain, it is told to run until a given “stop time,” usually the time of the events at the inputs to the internal domain that triggered the invocation. This “stop time” is the current time of the outer scheduler. Since the inner scheduler is requested to not exceed that time, it can only produce events with time stamp equal to that time. Thus, a timed domain wormhole, when fired, will always either produce no output events, or produce output events with time stamp equal to the simulated time at which it was fired.

To get a time delay through such a wormhole, two firings are required. Suppose the first firing is triggered by an input event at time T, then the inside system generates an internal event at a future time $T + \tau$. Before returning control to the outer scheduler, the inner scheduler requests that it be reinvoked at time $T + \tau$. When the “current time” of the outer scheduler reaches $T + \tau$, it reinvokes the inner scheduler, which then produces an output event at time $T + \tau$.

With this conservative style of timed interaction, we say that the DE domain operates in the *synchronized mode*. Synchronized mode operation suffers significant overhead at run time, since the wormhole is called at every time increment in the inner timed domain. Sometimes, however, this approach is too conservative.

In some applications, when an input event arrives, we can safely execute the wormhole into the future until either (a) we reach the time of the next event on the event queue of the outer domain, or (b) there are no more events to process in the inner domain. In other words, in certain situations, we can safely ignore the request from the output domain that we execute only up until the time of the input event. As an experimental facility to improve run-time efficiency, an option avoids synchronized operation. Then, we say that the DE domain operates in the *optimized mode*. We specify this mode by setting the target parameter *syncMode* to `FALSE` (zero). This should only be done by knowledgeable users who understand the DE model of computation very well. The default value of the *syncMode* parameter is `TRUE` (one), which means synchronized operation.

12.2.6 DE Performance Issues

DE Performance can be an issue with large, long-running universes. Below we discuss a few potential solutions.

The calendar queue scheduler is not always the one to use. It works well as long as the "density" of events in simulated time is fairly uniform. But if events are very irregularly spaced, you may get better performance with the simpler scheduler, because it makes no assumptions about timestamp values. For example, Tom Lane reported that the CQ scheduler did not behave well in a simulation that had a few events at time zero and then the bulk of the events between times 800000000 and 810000000 --- most of the events ended up in a single CQ "bucket", so that performance was worse than the simple scheduler.

Tom Lane also pointed out that both the CQ and simple schedulers ultimately depend on simple linear lists of events. If your application usually has large numbers of events pending, it might be worth trying to replace these lists with genuine priority queues (i.e., heaps, with O(log N) rather than O(N) performance). But you ought to profile first to see if that's really a time sink.

Another thing to keep in mind that the overhead for selecting a next event and firing a star is fairly large compared to other domains such as SDF. It helps if your stars do a reasonable amount of useful work per firing; that is, DE encourages "heavyweight" stars. One way to get around this is to put purely computational subsystems inside SDF wormholes. As discussedpreviously, an SDF-in-DE wormhole acts as a zero-delay star.

If you are running a long simulation, you should be sure that your machine is not paging or worse yet swapping; you should have plenty of memory. Usually 64Mb is enough, though 128Mb can help (`gdb` takes up a great deal of memory when you use it, too.). Depending on what platform you are on, you may be able to use the program `top` (`ftp://eecs.nwu.edu/pub/top`). You might also find it useful to use `iostat` to see if you are paging or swapping.

One way to gain a slight amount of speed is to avoid the GUI interface entirely by using `ptcl`, which does not have Tk stars. See "Some hints on advanced uses of ptcl with pigi" on page 3-19 for details.

12.3 An overview of stars in DE

The model of computation in the DE domain makes it amenable to high-level system modeling. For this reason, stars in the DE domain are often more complicated, and more specialized than those in the SDF domain. The stars that are distributed with the domain, therefore, should be viewed primarily as examples. They do not form a comprehensive set.

We have made every attempt to include in the distribution all of the reasonably generic stars that have been developed, plus a selection of the more esoteric ones (as examples). Keep in mind that the star libraries of the other domains are also available through the wormhole mechanism. Users that find themselves frequently needing stars from other domains may wish to build a library of single-star galaxies. Such galaxies can be used in any domain, regardless of the domain in which the single star resides. Ptolemy automatically implements this as a wormhole.

The top-level palette is shown in figure 12-4.

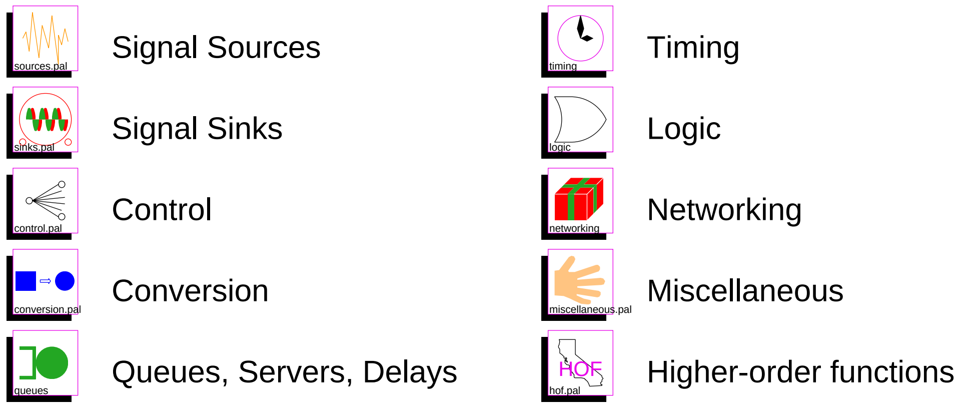


FIGURE 12-4: The top level palette of discrete-event stars.

The following star is available in all the palettes:

`BlackHole`	Discard all input particles.

12.3.1 Source stars

Strictly speaking, source stars are stars with no inputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. By convention, these stars are fired once at time zero automatically. During this and all subsequent firings, the star itself must determine when its next firing should occur. It schedules this next firing with a call to the method `refireAtTime(`*`time`*`)`. The source palette is shown in figure 12-5.

`Clock`	Generate events at regular intervals, starting at time zero.
`Impulse`	Generate a single event at time zero.
`Null`	Do nothing. This is useful for connecting to unused input ports.
`Poisson`	Generate events according to a Poisson process. The first event is generated at time zero. The mean inter-arrival time and magnitude of the events are given as parameters.
`PulseGen`	Generate events with specified values at specified moments. The events are specified in the *value* array, which consists of time-value pairs, given in the syntax of complex numbers.
`TclScript`	(Two icons.) Invoke a Tcl script. The script is executed at the start of the simulation, from within the star's begin method. It may define a procedure to be executed each time the star fires, which can in turn produce output events. There is a chapter of

	the Programmer's Manual devoted to how to write these scripts.
`TkButtons`	(Two icons.) Output the specified value when buttons are pushed. If the *allow_simultaneous_events* parameter is `YES`, the output events are produced only when the button labeled "PUSH TO PRODUCE EVENTS" is pushed. The time stamps of each output event is set to the current time of the scheduler when the button is pushed.
`TkSlider`	Output a value determined by an interactive on-screen scale slider.

For convenience, some stars are included in the source palette that are not really source stars, in the above sense. They require an input event in order to produce an output. These are listed below. The value of the input event is ignored; it is only its time stamp that matters.

`Const`	Produce an output event with a constant value (the default value is zero) when stimulated by an input event. The time stamp of the output is the same as that of the input.
`Ramp`	Produce an output event with a monotonically increasing value when stimulated by an input event. The value of the output event starts at *value* and increases by *step* each time the star fires.
`RanGen`	(Four icons.) Generate a sequence of random numbers. Upon receiving an input event, it generates a random number with `uniform`, `exponential`, or `normal` distribution, as determined by the *distribution* parameter. Depending on the distribu-

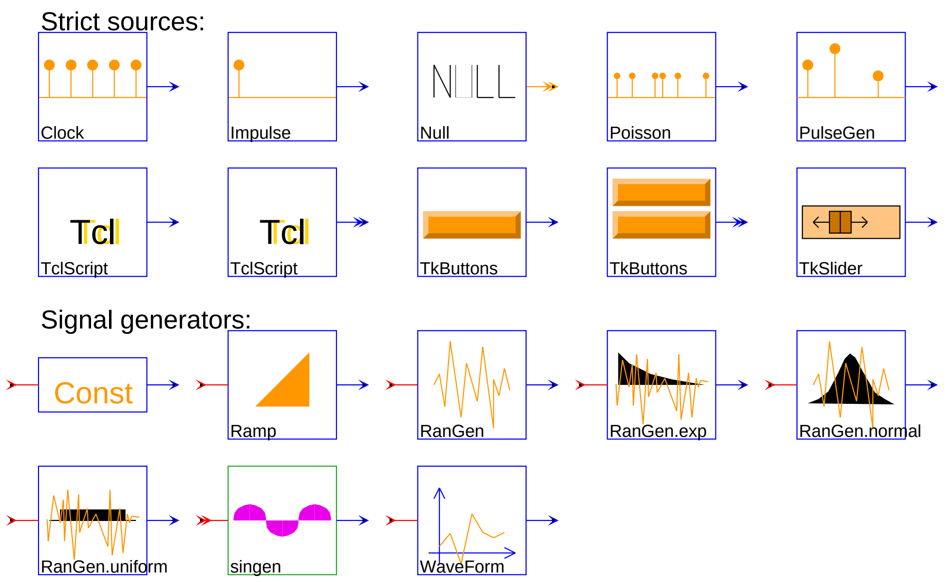


FIGURE 12-5: Source stars in the DE domain.

tion, other parameters specify either the mean and variance or the lower and upper extent of the range.

`singen` Generate a sample of a sine wave when triggered. This DE galaxy contains an SDF singen galaxy (i.e., a wormhole).

`WaveForm` Upon receiving an input event, output the next value specified by the array parameter *value* (default "1 -1"). This array can periodically repeat with any period, and you can halt a simulation when the end of the array is reached. The following table summarizes the capabilities:

haltAtEnd	*periodic*	*period*	*operation*
NO	**YES**	**0**	**The period is the length of the array**
NO	**YES**	**N>0**	**The period is N**
NO	**NO**	**anything**	**Output the array once, then zeros**
YES	**anything**	**anything**	**Stop after outputting the array once**

The first line of the table gives the default settings. The array may be read from a file by simply setting *value* to something of the form `< filename`.

12.3.2 Sink stars

The sink stars pointed to by the palette in figure 12-6 are those with no outputs. They display signals in various ways, or write them to files. Several of the stars in this palette are based on the `pxgraph` program. This program has many options, summarized in "pxgraph —

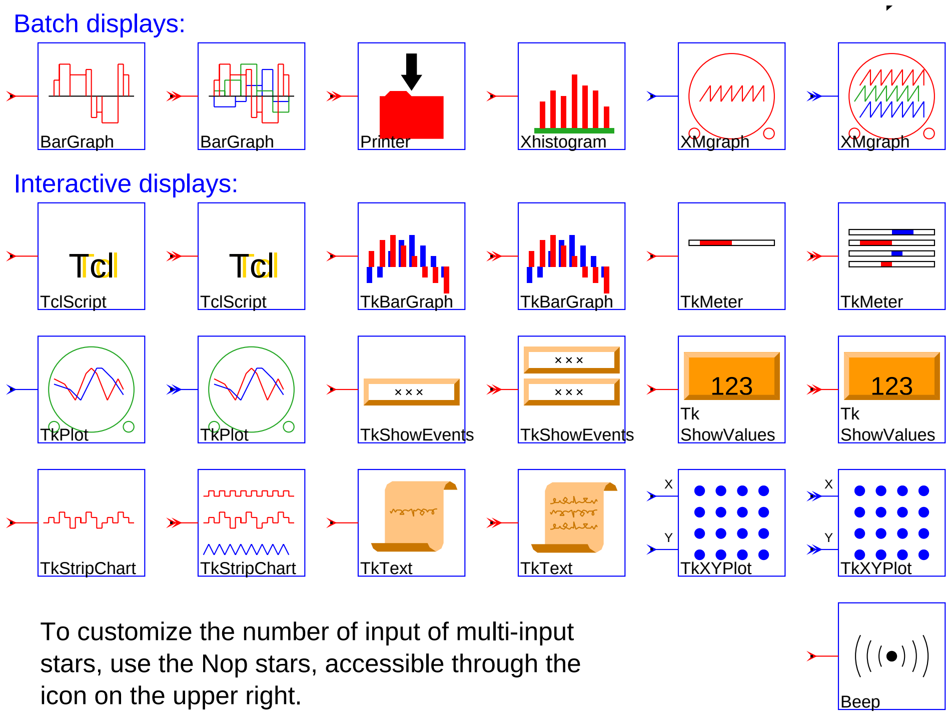


FIGURE 12-6: Sink stars in the DE domain.

The Plotting Program” on page 20-1. The differences between stars often amount to little more than the choice of default options. Some, however, preprocess the signal in useful ways before passing it to the `pxgraph` program. The first two icons actually correspond to only one star, with two different configurations. The first allows only one input signal, the second allows any number (notice the double arrow on the input port).

`BarGraph`	(Two icons.) Generate a plot with the `pxgraph` program that uses a zero-order hold to interpolate between event values. Two points are plotted for each event, one when the event first occurs, and the second when the event is supplanted by a new event. A horizontal line then connects the two points. If *draw_line_to_base* is `YES` then a vertical line to the base of the bar graph is also drawn for each event occurrence.
`Printer`	Print the value of each arriving event, together with its time of arrival. The *fileName* parameter specifies the file to be written; the special names `<stdout>` and `<cout>` (specifying the standard output stream), and `<stderr>` and `<cerr>` (specifying the standard error stream), are also supported.
`Xhistogram`	Generate a histogram with the `pxgraph` program. The parameter *binWidth* determines the width of a bin in the histogram. The number of bins will depend on the range of values in the events that arrive. The time of arrival of events is ignored. This star is identical to the SDF star `Xhistogram`, but is used often enough in the DE domain that it is provided here for convenience.
`XMgraph`	(Two icons.) Generate a plot with the `pxgraph` program with one point per event. Any number of event sequences can be plotted simultaneously, up to the limit determined by the `XGraph` class. By default, a straight line is drawn between each pair of events.
`TclScript`	(Two icons.) Invoke a Tcl script. The script is executed at the start of the simulation, from within the star’s begin method. It may define a procedure to be executed each time the star fires, which can in turn read input events. There is a chapter of the Programmer's Manual that explains how to write these scripts.
`TkBarGraph`	(Two icons.) Take any number of inputs and dynamically display their values in bar-chart form.
`TkMeter`	(Two icons.) Dynamically display the value of any number of input signals on a set of bar meters.
`TkPlot`	(Two icons.) Plot Y input(s) vs. time with dynamic updating. Retracing is done to overlay successive time intervals, as in an oscilloscope. The *style* parameter determines which plotting style is used: `dot` causes individual points to be plotted, whereas `connect` causes connected lines to be plotted. The *repeat_border_points* parameter determines whether rightmost

	events are repeated on the left. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.
`TkShowEvents`	(Two icons.) Display input event values together with the time stamp at which they occur. The print method of the input particles is used to show the value, so any data type can be handled, although the space allocated on the screen may need to be adjusted.
`TkShowValues`	(Two icons.) Display the values of the inputs in textual form. The print method of the input particles is used, so any data type can be handled, although the space allocated on the screen may need to be adjusted.
`TkStripChart`	(Two icons.) Display events in time, recording the entire history. The supported styles are `hold` for zero-order hold, `connect` for connected dots, and `dot` for unconnected dots. An interactive help window describes other options for the plot.
`TkText`	(Two icons.) Display the values of the inputs in a separate window, keeping a specified number of past values in view. The print method of the input particles is used, so any data type can be handled.
`TkXYPlot`	(Two icons.) Plot Y input(s) vs. X input(s) with dynamic updating. Time stamps are ignored. If there is an event on only one of a matching pair of X and Y inputs, then the previously received value (or zero if none) is used for the other. The *style* parameter determines which plotting style is used: `dot` causes individual points to be plotted, whereas `connect` causes connected lines to be plotted. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.
`Beep`	Cause a beep on the terminal when fired.

12.3.3 Control stars

Control stars (figure 12-7) manipulate the flow of tokens. All of these stars are polymorphic; they operate on any data type. From left to right, top to bottom, they are:

`Discard`	Discard input events that occur before the threshold time. Events after the threshold time are passed immediately to the output. This star is useful for removing transients and studying steady-state effects.
`Fork`	(Five icons.) Replicate input events on the outputs with zero delay.

`LossyInput`	Route inputs to the “sink” output with the probability *lossProbability* set by the user. All other inputs are sent immediately to the “save” output.
`Merge`	(Four icons.) Merge input events, keeping temporal order. Simultaneous events are merged in the order of the port number on which they appear, with port #1 being processed first.
`PassGate`	If the gate (bottom input) is open, then particles pass from “input” (left input) to “output.” When the gate is closed, no outputs are produced. If input particles arrive while the gate is closed, the most recent one will be passed to “output” when the gate is reopened.
`Router`	(Three icons.) Route an input event randomly to one of its outputs. The probability is equal for each output. The time delay is zero.
`Sampler`	Sample the input at the times given by events on the “clock” input. The data value of the “clock” input is ignored. If no input is available at the time of sampling, the latest input is used. If there has been no input, then a “zero” particle is produced. The exact meaning of zero depends on the particle type.
`LeakBucket`	Discard inputs that arrive too frequently. That is, any input event that would cause a queue of a given size followed by a server with a given service rate to overflow are discarded. Inputs that are not discarded are passed immediately to the output.

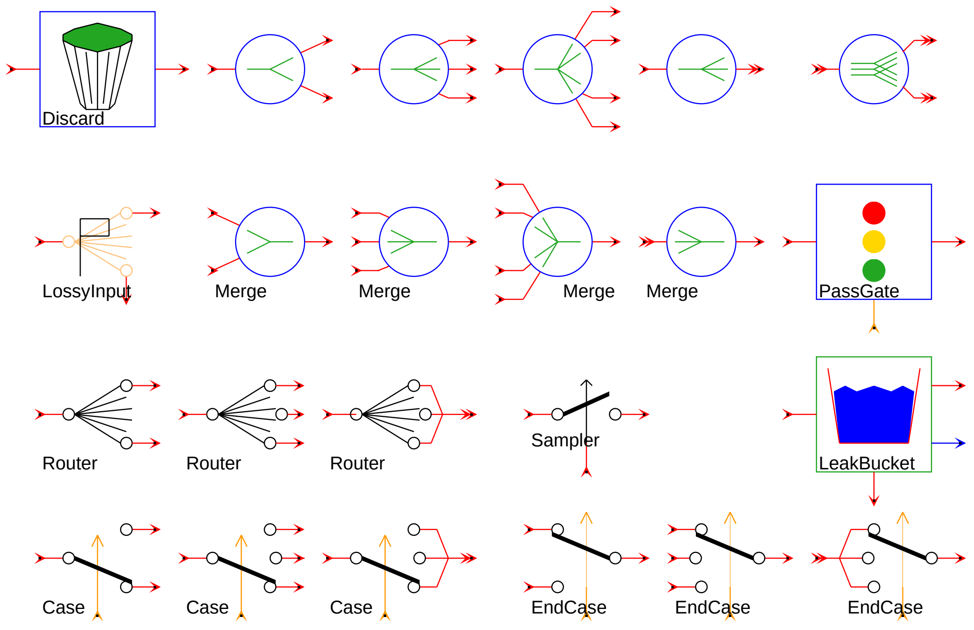


FIGURE 12-7: Control stars for the DE domain.

`Case`	(Three icons.) Switch input events to one of N outputs, as determined by the last received control input. The value of the control input must be between 0 and N-1, inclusive, or an error is flagged.
`EndCase`	(Three icons,) Select an input event from one of N inputs, as specified by the last received control input. The value of the control input must be between 0 and N-1 inclusive, or an error is flagged.

12.3.4 Conversion stars

The palette in figure 12-10 is intended to house a collection of stars for format conversions of various types. As of this writing, however, this collection is very limited. The first two stars in the conversion palette illustrate the consolidation of multiple data sample into single particles that can be transmitted as a unit. These stars use the class `FloatVecData`, which is simply a vector of floating-point numbers.

`Packetize`	Convert a number of floating-point input samples into a packet of type `FloatVecData`. A packet is produced when either an input appears on the demand input or when *maxLength* data values have arrived. Note that a null packet is produced if a demand signal arrives and there is no data.
`UnPacketize`	Convert a packet of type `FloatVecData` into a number of floating-point output samples. The "data" input feeds packets to the star. Whenever a packet arrives, the previous packet, if any, is discarded; any remaining contents are discarded. The "demand" input requests output data. If there is no data left in

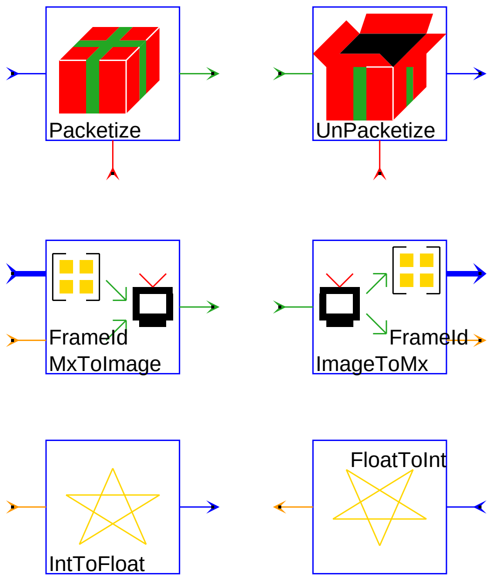


FIGURE 12-8: Type conversion stars for the DE domain.

	the current packet, the last output datum is repeated (zero is used if there has never been a packet). Otherwise the next data value from the current input packet is output.
`MxtoImage`	Convert a Matrix to a `GrayImage` output. The double values of the `FloatMatrix` are converted to the integer values of the `GrayImage` representation.
`ImageToMx`	Accept a black-and-white-image from an input image packet, and copy the individual pixels to a `FloatMatrix`. Note that even though the `GrayImage` input contains all integer values, we convert to a `FloatMatrix` to allow easier manipulation.

12.3.5 Queues, servers, and delays

The palette in figure 12-9 contains stars that model queues, servers, and time delays of various types. In the DE domain, the delay icon (the small green diamond at the upper left of the palette) does not represent a time delay. See "The DE target and its schedulers" on page 12-1.

`Delay`	Send each input event to the output with its time stamp incremented by an amount given by the *delay* parameter.
`VarDelay`	Delay the input by a variable amount. The *delay* parameter gives the initial delay, and the delay is changed using the "newDelay" input.
`PSServer`	Emulate a deterministic, processor-sharing server. If input events arrive when it is not busy, it delays them by the nominal service time. If they arrive when it is busy, the server is shared. Hence prior arrivals that are still in service will be delayed by more than the nominal service time.
`Server`	Emulate a server. If input events arrive when it is not busy, it delays them by the service time (a constant parameter). If they arrive when it is busy, it delays them the service time plus how-

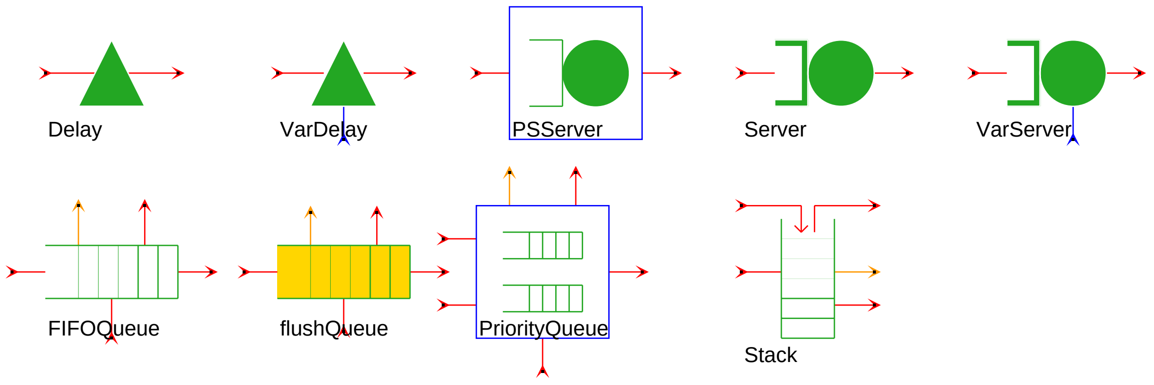


FIGURE 12-9: Queues, servers, and delays in the DE domain.

	ever long it takes to become free of previous tasks.
`VarServer`	Emulate a server with a variable service time. If input events arrive when it is idle, they will be serviced immediately and will be delayed only by the service time. If input events arrive while another event is being serviced, they will be queued. When the server becomes free, it will service any events waiting in its queue.
`FIFOQueue`	Implement a first-in first-out (FIFO) queue with finite or infinite length. Events on the “demand” input trigger a dequeue on the “outData” port if the queue is not empty. If the queue is empty, then a “demand” event enables the next future “inData” particle to pass immediately to “outData”. The first particle to arrive at “inData” is always passed directly to the output, unless *numDemandsPending* is initialized to 0. If *consolidateDemands* is set to `TRUE` (the default), then *numDemandsPending* is not permitted to rise above one. The size of the queue is sent to the *size* output whenever an “inData” or “demand” event is processed. Input data that doesn't fit in the queue is sent to the “overflow” output.
`FlushQueue`	Implement a FIFO queue that when full, discards all inputs until it empties completely.
`PriorityQueue`	Emulate a priority queue. Inputs have priorities according to the number of the input, with “inData#1” having highest priority, “inData#2” being next, etc. When a “demand” is received, outputs are produced by selecting first based on priority, and then based on time of arrival, using a FIFO policy. A finite total capacity can be specified by setting the *capacity* parameter to a positive integer. When the capacity is reached, further inputs are sent to the “overflow” output, and not stored. The *numDemandsPending* and *consolidateDemands* parameters have the same meaning as in other queue stars. The size of the queue is sent to the “size” output whenever an “inData” or “demand” event is processed.
`Stack`	Implement a stack with either finite or infinite length. Events on the “demand” input pop data from the stack to “outData” if the stack is not empty. If it is empty, then a “demand” event enables the next future “inData” particle to pass immediately to “outData.” By default, *numDemandsPending* is initialized to 1, so the first particle to arrive at “inData” is passed directly to the output. If *consolidateDemands* is set to `TRUE` (the default), then *numDemandsPending* is not permitted to rise above one. The size of the stack is sent to the “size” output whenever an “inData” or “demand” event is processed. Input data that doesn’t fit on the stack is sent to the “overflow” output.

The following star does not appear in the palette.

`QueueBase`	Serve as the base class for FIFO and LIFO queues. This star is not intended to be used except to derive useful stars. All inputs are simply routed to the “overflow” output. None are stored.

12.3.6 Timing stars

The palette in figure 12-10 contains stars that are primarily concerned with either simulated or real time.

`MeasureDelay`	Measure the time difference between the first arrival and the second arrival of an event with the same value. The second arrival and the time difference are each sent to outputs.
`MeasureInterval`	The value of each output event is the simulated time since the last input event (or since zero, for the first input event). The time stamp of each output event is the time stamp of the input event that triggers it.
`StopTimer`	Generate an output at the `stopTime` of the `DEScheduler` under which this block is running. This can be used to force actions at the end of a simulation. Within a wormhole, it can used to force actions at the end of each invocation of the wormhole. An input event is required to enable the star.
`Timeout`	Detect time-out conditions and generate an alarm if too much time elapses before resetting or stopping the timer. Events arriving on the “Set” input reset and start the timer. Events arriving on the “Clear” input stop the timer. If no “Set” or “Clear” events arrive within *timeout* time units of the most recent “Set”, then that “Set” event is sent out the “alarm” output.
`TimeStamp`	The value of the output events is the time stamp of the input events. The time stamp of the output events is also the time stamp of the input events.

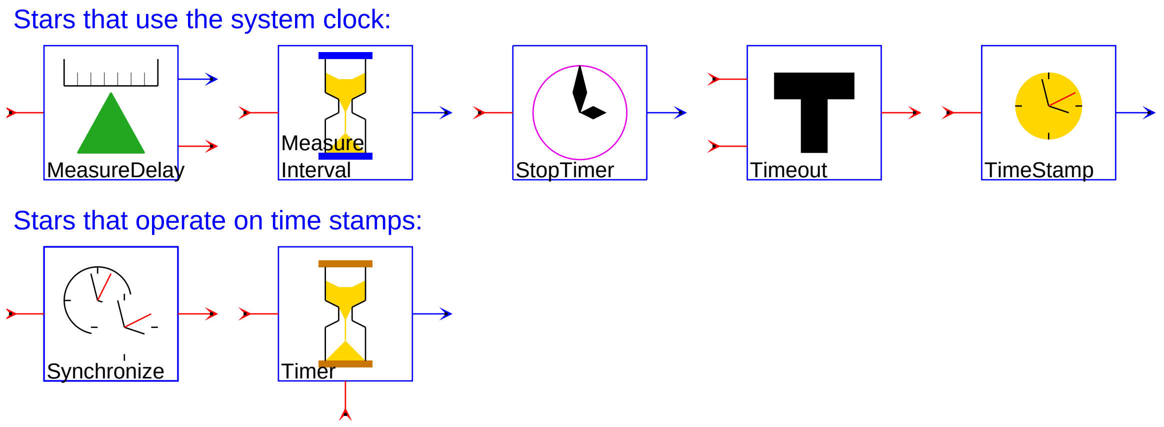


FIGURE 12-10: Timing stars for the DE domain.

`Synchronize`	Hold input events until the time elapsed on the system clock since the start of the simulation is greater than or equal to their time stamp. Then pass them to the output.
`Timer`	Upon receiving a trigger input, output the elapsed real time in seconds, divided by *timeScale*, since the last reset input, or since the start of the simulation if no reset has been received. The time stamp of the output is the same as that of the trigger input. The time in seconds is related to the scheduler (simulated) time through the scaling factor `timeScale`.

The following star does not appear in the palette, because it is not intended to be used directly in Ptolemy applications.

`TimeoutStar`	Serve as the base class for stars that check time-out conditions. The methods "set", "clear", and "expired" are provided for setting and testing the timer.

12.3.7 Logic stars

The logic palette in figure 12-10 is made up of only three stars, with the multiplicity of icons representing different configurations of these stars.

`Test`	(Six icons) Compare two inputs. The test condition can be any of {`EQ NE LT LE GT GE`} or {`== != < <= > >=`}, resulting in equals, not equals, less-than, less-than or equals, etc. If *crossingsOnly* is `TRUE`, then an output event is generated only when the value of the output changes. Hence the output events will always alternate between true and false.
`Logic`	(Nineteen Icons) Apply a logical operation to any number of

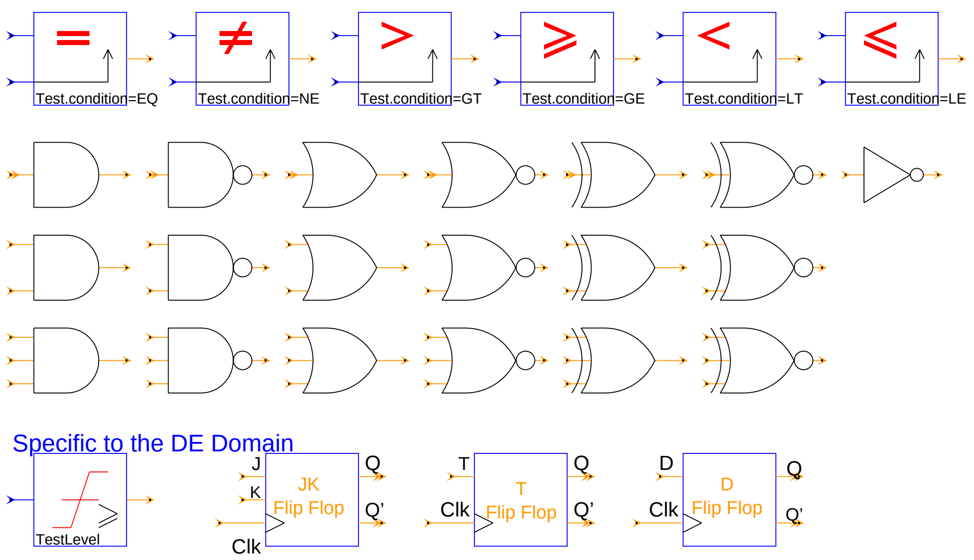


FIGURE 12-11: Logic stars for the DE domain.

	inputs. The inputs are integers interpreted as Booleans, where zero is a `FALSE` and nonzero is a `TRUE`. The logical operations supported are {`NOT AND NAND OR NOR XOR XNOR`}.
`TestLevel`	Detect threshold crossings if the *crossingsOnly* parameter is `TRUE`. Otherwise, it simply compares the input against the “threshold.” If *crossingsOnly* is `TRUE`, then: (1) a `TRUE` is sent to “output” when the “input” particle exceeds or equals the “threshold” value, having been previously smaller; (2) a `FALSE` is sent when the “input” particle is smaller than “threshold” having been previously larger. Otherwise, no output is produced. If *crossingsOnly* is `FALSE`, then a `TRUE` is sent to “output” whenever any “input” particle greater than or equal to “threshold” is received, and a `FALSE` is sent otherwise.
`FlipFlop Stars`	Binary state is afforded in the DE logic palette with the inclusion of flip flop circuits. Three synchronous sequential circuit components, FlipFlopJK, FlipFlopT and FlipFlopD, serve as basic memory elements.

12.3.8 Networking stars

The palette shown in figure 12-12 includes stars that have been designed to model communication networks. These are illustrative of a common use of the DE domain, for modeling packet-switched networks. However, many of the stars are specialized to a particular type of network design. Thus, they should be viewed as illustrative examples, rather than as a comprehensive library.

A `NetworkCell` class is used in many of these stars. It models packetized data that is transmitted through cell-relay networks. Each `NetworkCell` object can carry any user data of type `Message`. In addition to this user data, the `NetworkCell` contains a destination address and a priority. These are used by stars and galaxies to route the cell through the network. The definition of the `NetworkCell` class may be found in `$PTOLEMY/src/domains/sdf/image/kernel`, since it is used in the SDF and DE domains, and was developed primarily for modeling packet-switched video.

Cell creation and access

`CellLoad`	Read in an `Envelope`, extract its `Message`, and output that `Message` in a `NetworkCell`. Append a destination and priority to the packet.
`CellUnload`	Remove a `Message` from a `NetworkCell`.
`ImageToCell`	Packetize an image. Each image is divided up into chunks no larger than *CellSize*. Each cell is delayed from its predecessor by *TimePerCell*. If a new input arrives while an older one is being processed, the new input is queued.

`CellToImage`	Read `NetworkCell` packets containing image data and output whole images. The current image is sent to the output when the star reads image data with a higher frame id than the current image. For each frame, the fraction of input data that was lost is sent to the "lossPct" output.

Cell routing, control, and service

`CellRoute`	Read `NetworkCell` packets from multiple input sources and route them to the appropriate output using a routing table that maps addresses into output ports.
`PriorityCheck`	Read `NetworkCell` packets from multiple input sources. If the priority of an input `NetworkCell` is less than the most recent "priority" input, then the cell is sent to the "discard" output. Otherwise it is sent to the "output" port.
`Switch4x4`	Implement a four-input, four-output network switch that can process objects of type `NetworkCell`, or any type derived from `NetworkCell`. Each `NetworkCell` object contains a destination address. This galaxy uses the destination address as an index into its *Routes* array parameter to choose an output

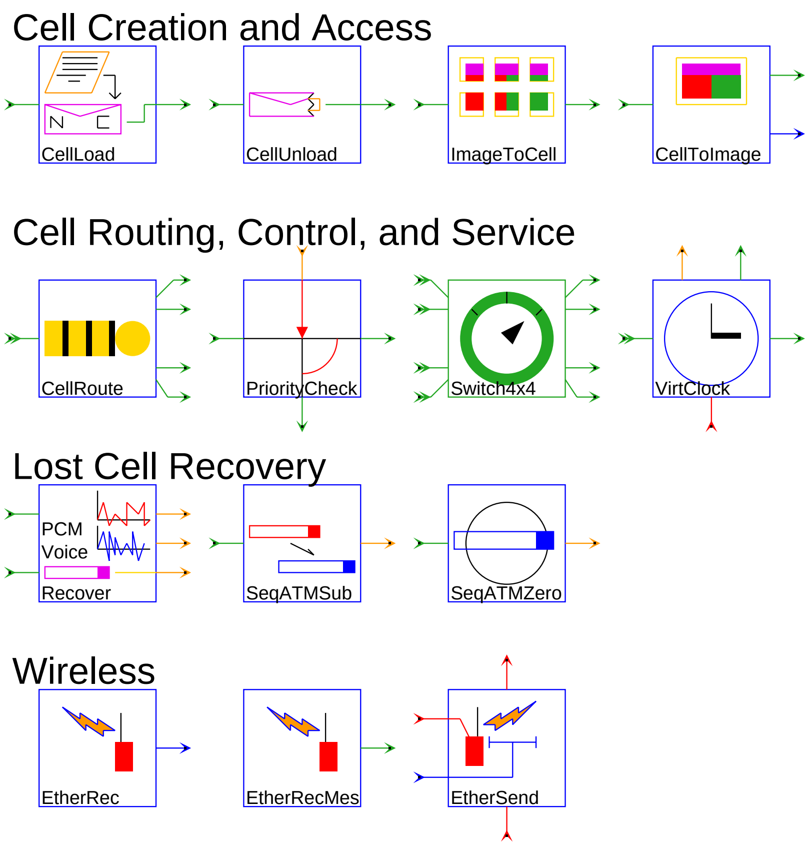


FIGURE 12-12: A palette of DE stars dedicated to modeling of communication networks.

port over which the input object will leave. A prioritized queueing scheme is used.

`VirtClock` Read a `NetworkCell`. It identifies which virtual circuit number the cell belongs to and then computes the virtual time stamp for the cell by applying the virtual clock algorithm (see the source code in $PTOLEMY/src/domains/de/stars/DEVirtClock.pl). It then outputs all cells in order of increasing virtual time stamp.

Upon receiving a “demand” input, the cell with the smallest time stamp is output. An output packet is generated for every demand input unless all of the queues are empty. Demand inputs arriving when all queues are empty are ignored.

The number of stored cells is output after the receipt of each “input” or “demand.”

When a cell arrives and the number of stored cells equals *MaxSize* then the cell with the biggest virtual time stamp is discarded. This cell may or may not be the new arrival. If *MaxSize* is zero or negative, then infinitely many cells can be stored.

Lost cell recovery

The stars in this subgroup implement a variety of mechanisms for replacing lost cells in a packet-switched network. They use a class called `SeqATMCell` that is designed to model packets in the proposed broadband integrated services digital network (BISDN). The class is derived from `Message`, but has added facilities for marking the packet with a sequence number, and setting and reading individual bits. The sequence number is used to determine when packets have been lost.

`PCMVoiceRecover` Input a stream of `SeqATMCell` objects. All the information bits in objects received with correct sequence numbers are sent to “output.”

If a missing `SeqATMCell` object is detected, this star sends the most recent 8 * *tempSize* received bits to the “temp” output, and the most recent (8 * *searchWindowSize* + *numInfoBits*) received bits to the “window” output.

The bits output on the “window” and “temp” outputs can be used by the `PatternMatch` galaxy to implement lost-speech recovery.

`SeqATMSub` Read a sequence of `SeqATMCells`. It will check sequence numbers, and if a `SeqATMCell` is found missing, the information bits of the previously arrived `SeqATMCell` will be output in its place.

The information bits from each correctly received `SeqATMCell` are unloaded and sent to the output port.

`SeqATMZero`	Read a sequence of `SeqATMCell` objects. For each object input correctly in sequence, *headerLength* bits are skipped over and the next *numInfoBits* bits in the cell are output. If this star finds, by checking sequence numbers, that a `SeqAT-MCell` is missing, it will substitute *numInfoBits* 0-bits for the missing bits.

Wireless network simulation

`Ether`	(Not shown in the palette.) This is the base class for transmitter and receiver stars that communicate over a shared medium. Each transmitter can communicate with any or all receivers that have the same value for the “medium” parameter. The communication is accomplished without graphical connections, and the communication topology can be continually changing. This base class implements the data structures that are shared between the transmitters and receivers.
`EtherRec`	Receive floating-point particles transmitted to it by an `EtherSend` star. The particle is produced at the output after some duration of transmission specified at the transmitter.
`EtherRecMes`	See the explanation for the `EtherRec` star.The only difference is that this stars forces the output to be a message.
`EtherSend`	Transmit particles of any type to any or all receivers that have the same value for the *medium* parameter. The receiver address is given by the “address” input, and it must be an string. If the string begins with a dash “`-`”, then it is interpreted as a broadcast request, and copies of the particle are sent to all receivers that use the same medium. The transmitter “occupies” the medium for the specified duration. A collision occurs if the medium is occupied when a transmission is requested. In this case, the data to be transmitted is sent to the “collision” output.

12.3.9 Miscellaneous stars

These stars are shown in figure 12-12.

Hardware modeling

`Arbitrate`	Act as a non-preemptive arbitrator, granting requests for exclusive control. If simultaneous requests arrive, priority is given to port A. When control is released, any pending requests on the other port will be serviced. The “requestOut” and “grantIn” connections allow interconnection of multiple arbitration stars for more intricate control structures.
`HandShake`	Cooperate with a possibly preemptive arbitrator through the

	"request" and "grant" controls. "Input" particles are passed to "output", and an "ackIn" particle must be received before the next "output" can be sent. This response is made available on "ackOut."
`handShakeQ`	Handshake with queued input events.
`TclScript`	Invoke a Tcl script. The script is executed at the start of the simulation, from within the star's begin method. It may define a procedure to be executed each time the star fires, which can in turn read input events and produce output events. There is a chapter of the Programmer's Manual devoted to how to write these scripts.

Statistics and monitoring

`Statistics`	Calculate the average and variance of the input values that have arrived since the last reset. An output is generated when a "demand" input is received. When a "reset" input arrives, the calculations are restarted. When "demand" and "reset" particles arrive at the same time, an output is produced before the calculations are restarted.
`UDCounter`	Implement an up/down counter. The processing order of the ports is: countUp -> countDown -> demand -> reset. Specifically, all simultaneous "countUp" inputs are processed. Then all simultaneous "countDown" inputs are processed. If there are multiple simultaneous "demand" inputs, all but the first are ignored. Only one output will be produced.

Signal processing

`Filter`	Filter the input signal with a first-order, autoregressive (AR)

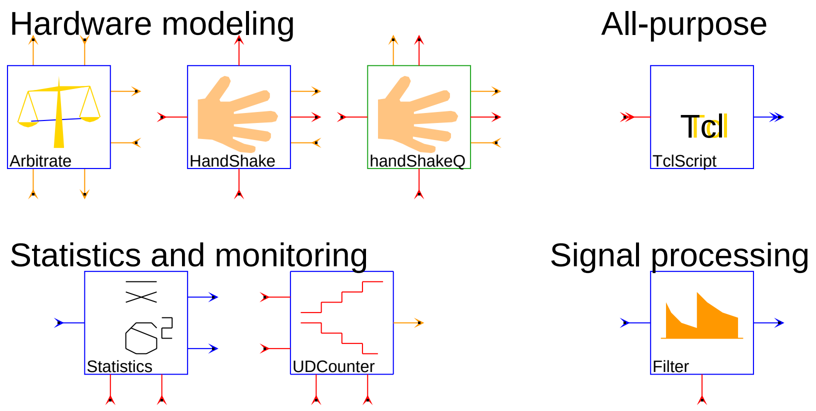


FIGURE 12-13: A palette of miscellaneous DE stars.

impulse response. The data input is interpreted as weighted impulses (Dirac delta functions). An output is triggered by a clock input.

12.3.10 HOF Stars

For a discussion of the HOF stars, please see the “An overview of the HOF stars” on page 6-15.

12.4 An overview of DE demos

The number of DE demos is considerably smaller than SDF. Many of the demos, however, are much more complex, often incorporating SDF subsystems to accomplish audio or video encoding. The top-level palette for demos in the discrete-event domain is shown in figure 12-14. The subpalettes are described below.

12.4.1 Basic demos

These demos illustrate the use of certain stars without necessarily performing functions that are particularly interesting. The palette is shown in figure 12-15. The individual demos are summarized below.

`caseDemo`	Demonstrates the `Case` star by deconstructing a Poisson counting process into three subprocesses.
`conditionals`	Demonstrate the use of the `Test` block in its various configurations to compare the values of input events with floating-point values. The input test signal is a pair of ramps, with each event repeated once after some delay. Since the ramps have different steps, they will cross.
`logic`	Demonstrate the use of the `Logic` star in its various instantia-

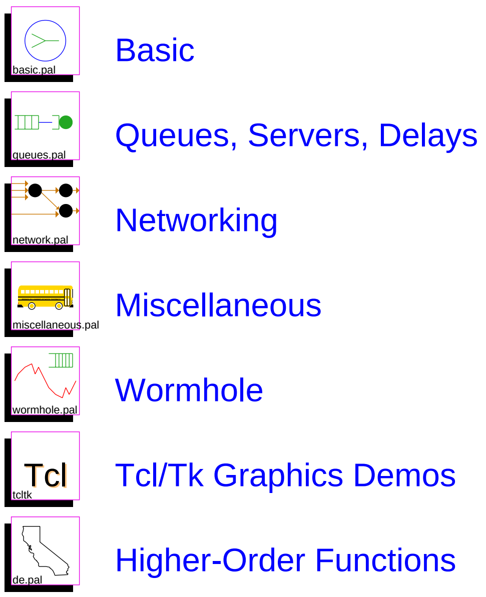


FIGURE 12-14: The top-level palette for DE demos.

	tions as AND, NAND, OR, NOR, XOR, XNOR and inverter gates. The three test signals consist of square waves with periods 2, 4, and 6.
`merge`	Demonstrate the `Merge` star. The star is fed two streams of regular arrivals, one a ramp beginning at 10.0, and one a ramp beginning at 0.0. The two streams are merged into one, in chronological order, with simultaneous events appearing in arbitrary order.
`realTime`	Demonstrate the use of the `Synchronize` and `Timer` blocks. Input events from a `Clock` star are held by the `Synchronize` star until their time stamp, multiplied by the universe parameter *timeScale*, is equal to or larger than the elapsed real time since the start of the simulation. The `Timer` star then measures the actual (real) time at which the `Synchronize` output is produced. The closer the resulting plot is to a straight line with a slope of one, the more precise the timing of the `Synchronize` outputs are.
`router`	Randomly route an irregular but monotonic signal (a Poisson counting process) through two channels with random delay, and merge the channel outputs.
`sampler`	Demonstrate the `Sampler` star. A counting process with regular arrivals at intervals of 5.0 is sampled at regular intervals of 1.0. As expected, this produces 5 samples for each level of the counting process.
`statistics`	Compute the mean and variance of a random process using the `Statistics` star. The mean and variance are sent to the stan-

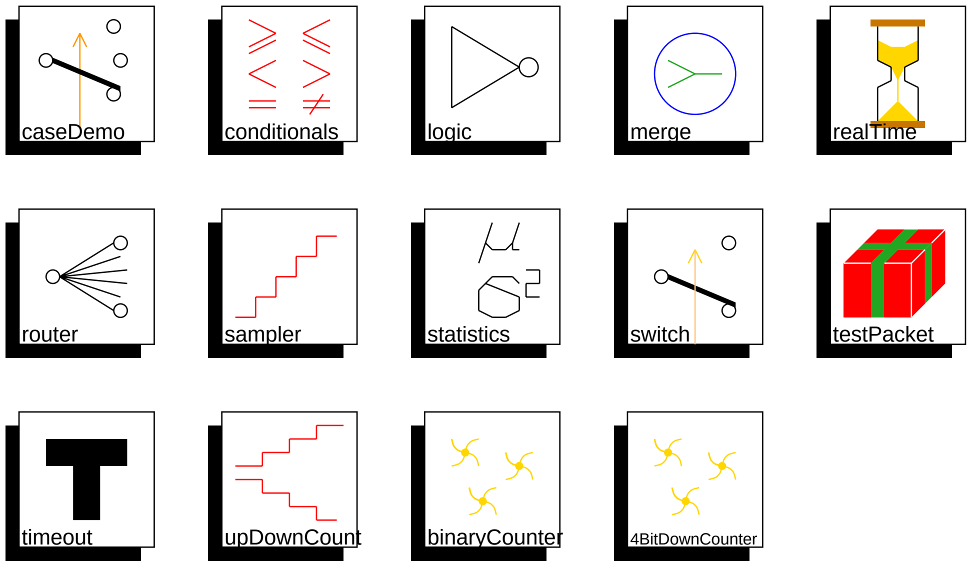


FIGURE 12-15: Basic DE demos.

	dard output when the simulation stops. This action is triggered by an event produced by the `StopTimer` star.
`switch`	Demonstrate the use of the `Switch` star. A Poisson counting process is sent to one output of the switch for the first 10 time units, and to the other output of the switch for the remaining time.
`testPacket`	Construct packets consisting of five sequential values from a ramp, send these packets to a server with a random service time, and then deconstruct the packets by reading the items in the packet one by one.
`timeout`	Demonstrate the use of the `Timeout` star. Every time unit, a timer is set. If after another 0.5 time units have elapsed, the timer is not cleared, an output is produced to indicate that the timer has expired. The signal that clears the timer is a Poisson process with a mean inter-arrival time of one time unit.
`upDownCount`	Demonstrate the `UDCounter` star. Events are generated at two different rates to count up and down. The up rate is faster than the down rate, so the trend is upwards. The value of the count is displayed every time it changes.
`binaryCounter`	Demonstrate the `FlipFlopJK` star.
`4BitDownCounter`	Demonstrate the use of the other Flip Flop stars.

12.4.2 Queues, servers, and delays

The palette of demos illustrating queueing systems is shown in figure 12-16. It includes:

`blockage`	Demonstrate a blocking strategy in a queueing network. In a cascade of two queues and servers, when the second queue fills up, it prevents any further dequeueing of particles from the first queue until it once again has space.
`delayVsServer`	Illustrate the difference between the `Delay` and `Server` blocks. The `Delay` passes the input events to the output with a

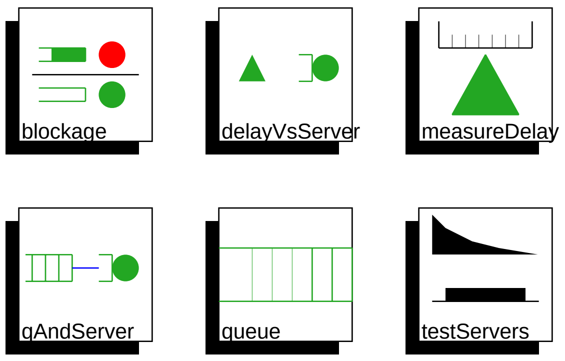


FIGURE 12-16: Queueing system demos

	fixed time offset. The `Server` accepts inputs only after the previous inputs have been served, and then holds that input for a fixed offset.
`measureDelay`	Demonstrate the use of the `MeasureDelay` block to measure the sojourn time of particles in a simple queueing system with a single server with a random service time.
`priority`	Demonstrate the use of the `PriorityQueue` block together with a `Server`. The upper input to the `PriorityQueue` has priority over the lower input. Thus, when the queue overflows, data is lost from the lower input.
`psServer`	Demonstrate the processor-sharing server. Unlike other servers, this server accepts new inputs at any time, regardless of how busy it is. Accepting a new input, however, slows down the service to all particles currently being served.
`qAndServer`	Demonstrate the use of the `FIFOQueue` and `Stack` stars together with `Servers`. A regular counting process is enqueued on both stars. The particles are dequeued whenever the server is free. The `Stack` is set with a larger capacity than the `FIFOQueue`, so it overflows second. Overflow events are displayed.
`queue`	Demonstrate the use of the `FIFOQueue` and `Stack` stars. A Poisson counting process is enqueued on both stars, and is dequeued at a regular rate, every 1.0 time units. The output of the `FIFOQueue` is always monotonically increasing, because of the FIFO policy, but the output of the `Stack` need not be. The Stack is set with a smaller capacity than the `FIFOQueue`, so it overflows first. Overflow events are displayed.
`testServers`	Demonstrate servers with random service times (uniform and exponential).

12.4.3 Networking demos

A major application of the DE domain is the simulation of communication networks. The palette in figure 12-17 contains such network simulations. The demos are:

`FlushNet`	Simulate a queue with "input flushing" during overflow. If the queue reaches capacity, all new arrivals are discarded until all

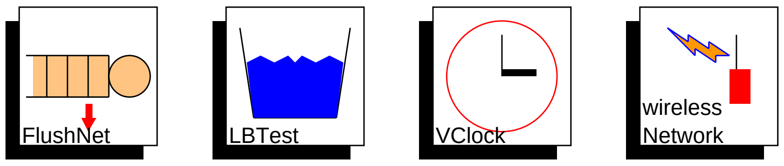


FIGURE 12-17: Networking demos

	items in the queue have been served.
`LBTest`	Simulate leaky bucket network rate controllers. These controllers moderate the flow of packets to keep them within specified rate and burstiness bounds.
`VClock`	Model a network with four inputs and virtual clock buffer service.
`wirelessNetwork`	Demonstrate shared media communication without graphical connectivity, using `EtherSend` and `EtherRec` stars. Two clusters on the left transmit to two clusters on the right over two distinct media, radio and infrared. The communication is implemented using shared data structures between the stars.

12.4.4 Miscellaneous demos

The palette in figure 12-18 shows miscellaneous demos. The first two of these model continuous-time random processes, although only discrete-time samples of these processes can be displayed.

`shotNoise`	Generate a continuous-time shot-noise process and display regularly spaced samples of it. The shot noise is generated by feeding a Poisson process into a `Filter` star.
`hdShotNoise`	Generate a high-density shot noise process and verify its approximately Gaussian distribution by displaying a histogram.

The following demos illustrate the use of the DE domain for high-level modeling of protocols for sharing hardware resources.

`roundRobin`	Simulate shared memory with round-robin arbitration at a high level.
`prioritized`	Simulate a shared memory with prioritized arbitration at a high level.

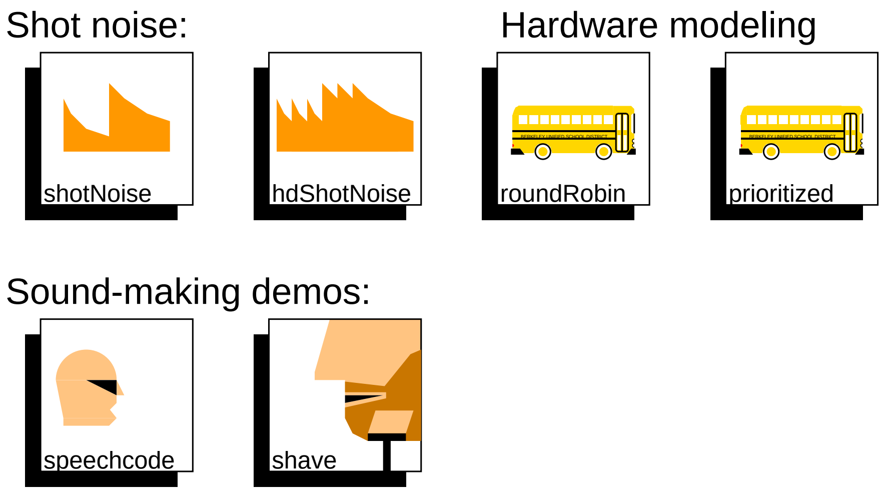


FIGURE 12-18: Miscellaneous demos.

The following demos make sounds.

`speechcode`	Perform speech compression with a combination of silence detection, adaptive quantization, and adaptive estimation. After speech samples are read from a file, they are encoded, packetized, depacketized, decoded, and played on the workstation speaker.
`shave`	Demonstrate the `Synchronize` star to generate a beeping sound with a real-time rhythm.

12.4.5 Wormhole demos

The palette in figure 12-19 shows some simple demonstrations of multiple domain simulations. Each of these combines SDF with DE. The demos are:

`distortion`	Show the effects on real-time signals of a highly simplified packet-switched network. Packets can arrive out of order, and they can also arrive too late to be useful. In this simplified system, a sinusoid is generated in the SDF domain, launched into a communication network implemented in the DE domain, and compared to the output of the communication network. Plots are given in the time and frequency domains of the sinusoid before and after the network.
`distortionQ`	Similar to the distortion demo. The only difference is in the `reorderQ` wormhole, which introduces queueing.
`worm`	Show how easy it is to use SDF stars to perform computation on DE particles. A Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples.
`four_level`	A four level SDF/DE/SDF/DE system.

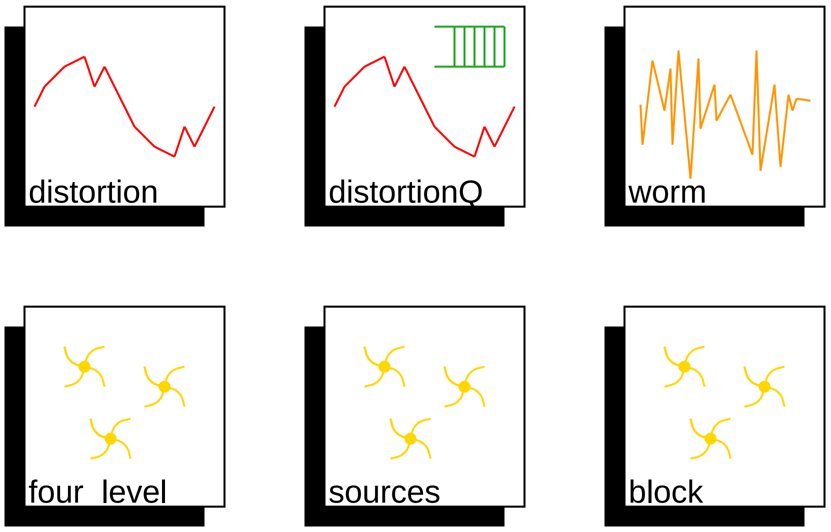


FIGURE 12-19: Wormhole demos

`sources`	Show how to use an SDF star as a source by using a dummy input into the SDF system. The SDF subsystem fires instantaneously from the perspective of DE. The *schedulePeriod* SDF target parameter has no effect.
`block`	The *schedulePeriod* parameter of the SDF target determines how the inside of the DE system interprets the timing of events arriving from SDF. When several samples are produced in one iteration, as here, the time stamps of the corresponding events are uniformly distributed over the schedule period.

12.4.6 Tcl/Tk Demos

The palette in figure 12-20 contains the Tcl/Tk demos

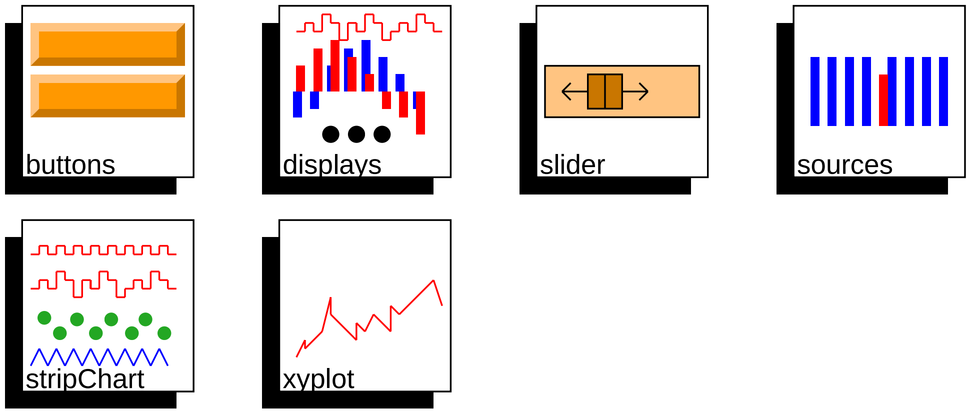


FIGURE 12-20: Tcl/Tk DE demos

`buttons`	Demonstrate `TkButtons` by having the buttons generate events asynchronously with the simulation.
`displays`	Demonstrate some of the interactive displays in the DE domain.
`slider`	Demonstrate `TkSlider` by having the slider produce events asynchronously. The asynchronous events are plotted together with a clock, which produces periodic outputs in simulated time. Notice that the behavior is roughly the same regardless of the interval of the clock.
`sources`	A Tcl script writes asynchronously to its output roughly periodically in real time (using the Tk “after” command). The asynchronous events are plotted together with a clock, which produces periodic outputs in simulated time. Notice that the plot looks roughly the same regardless of the interval of the clock.
`stripChart`	Demonstrate the `TkStripChart` by plotting several different

	sources.
`xyplot`	Display queue size as a function of time with an exponential random server. Note that the `TkPlot` star overlays the plots as time progresses, which the `TkXYPlot` star does not. Thus, the points on the `TkXYPlot` star go off the screen to the right. The `TkStripChart` star records the entire history.

12.4.7 HOF Demos

For information on the HOF demos, see “HOF demos in the DE domain” on page 6-20.

Chapter 13. CG Domain

Authors: *Joseph T. Buck*
Soonhoi Ha
Christopher Hylands
Edward A. Lee
Praveen Murthy
Thomas Parks
José Luis Pino
Kennard White

13.1 Introduction

The Code Generation (CG) domain and its derivative domains, such as the CG56 domain (Motorola DSP56000) and the C language (CGC) domain, are used to generate code rather than to run simulations. Only the derivative domains are of practical use for generating code. The stars in the CG domain itself can be thought of as “comment generators”; they are useful for testing and debugging schedulers and for little else. The CG domain is intended as a model and a collection of base classes for derivative domains. This section documents the common features and general structure of all code generation domains.

All the code generation domains that are derived from the CG domain in this release obey SDF semantics and can thus be scheduled at compile time. Internally, however, the CG only assumes that stars obey data flow semantics. Currently, we have implemented two approaches for data-dependent execution, CGDDF, which recognizes and implements certain commonly used programming constructs [Sha92], and BDF (“Boolean dataflow” or the token-flow model) [Buc93c]. Even when these are implemented, the vast majority of stars in any given application should obey the SDF rules to permit efficient multiprocessor code generation.

A key feature of code generation domains is the notion of a target architecture. Every application must have a user-specified target architecture, selected from a set of targets supported by the user-selected domain. Every target architecture is derived from the base class `Target`, and controls such operations as scheduling, compiling, assembling, and downloading code. Since the target controls scheduling, multiprocessor architectures can be supported with automated task partitioning and synchronization.

Another feature of the code generation domains is the ability to use different schedulers. A key idea in Ptolemy is that there is no single scheduler that is expected to handle all situations. We have designed a suite of specialized schedulers that can be mixed and matched for specific applications. Some targets in the CG domain, in addition to serving as base classes for derived domains, allow the user to experiment with these various schedulers.

13.2 Targets

A code generation Domain is specific to the language generated, such as C (CGC) and

DSP56000 assembly code (CG56). In previous versions of Ptolemy, we released code generation domains for the Sproc assembly language [Mur93], the DSP96000 assembly language, and the Silage language. Each code generation domain has a default target which defines routines generic to the target language. These targets are derived from targets defined in the CG domain.

A `Target` object has methods for generating a schedule, compiling the code, and running the code (which may involve downloading code to the target hardware and beginning its execution). There also may be child targets (for representing multiprocessor targets) together with methods for scheduling the communication between them. Targets also have parameters that are user specified. There are four targets in the CG domain; these are described below.

13.2.1 default-CG

This target is the default target for the CG domain. It allows the user to experiment with the various uniprocessor schedulers. Currently, there is a suite of schedulers that generate schedules of various forms of optimality. For instance, the default SDF scheduler generates schedules that try to minimize the amount of buffering required on arcs, while the loop schedulers try to minimize the amount of code that is generated. Refer to the schedulers section in this chapter for a discussion on these schedulers. There are only two parameters for this target:

directory	(`STRING`) Default = `$HOME/PTOLEMY_SYSTEMS` This is the directory to which all generated files will be written to.
looping Level	(`STRING`) Default = `ACYLOOP` The choices are DEF, CLUST, SJS, or ACYLOOP. Case does not matter; ACYLOOP is the same as AcyLoOP. If the value is DEF, no attempt will be made to construct a looped schedule. This can result in very large programs for multirate systems, since inline code generation is used, where a codeblock is inserted for each appearance of an actor in the schedule. Setting the level to CLUST invokes a quick and simple loop scheduler that may not always give single appearance schedules. Setting it to SJS invokes the more sophisticated SJS loop scheduler [Bha93c], which can take more time to execute, but is guaranteed to find single appearance schedules whenever they exist. Setting it to ACYLOOP invokes a scheduler that generates single appearance schedules optimized for buffer memory usage [Mur96][Bha96], as long as the graph is acyclic. If the graph is not acyclic, and ACYLOOP has been chosen, then the target automatically reverts to the SJS scheduler. For backward compatibility, “0” or “NO”, “1”, and “2” or “YES” are also recognized, with “0” or “NO” being DEF, “1” being CLUST, and “2” or “YES” being SJS. NOTE: Loop scheduling only applies to uniprocessor targets; hence, this parameter does not appear in the `FullyConnected` target.

In addition to these parameters, there are a number of parameters that are in this target that are not visible to the user. These parameters may be made visible to the user by derived targets. The complete list of these parameters follows:

host	(`STRING`) Default = The default is the empty string. This is the host machine to compile or assemble code on. All code is written to and compiled and run on the computer specified by this parameter. If a remote computer is specified here then `rsh` commands are used to place files on that computer and to invoke the compiler. You should verify that your .rhosts file is properly configured so that `rsh` will work.
file	(`STRING`) Default = The default is the empty string. This represents the prefix for file names for all generated files.
display?	(`INT`) Default = `YES` If this flag is set to `YES`, then the generated code will be displayed on the screen.
compile?	(`INT`) Default = `YES` If this flag is set to `YES`, then the generated code will be compiled (or assembled).
load?	(`INT`) Default = `YES` If this flag is set to `YES`, then the compiled code will be loaded onto a chip.
run?	(`INT`) Default = `YES` If this flag is set to `YES`, then the generated code is run.

13.2.2 bdf-CG

This target demonstrates the use of BDF semantics in code generation. It uses the BDF scheduler to generate code. See the BDF domain documentation for more information on BDF scheduling. There is only one target parameter available to the user; the `directory` parameter above. This parameter has the same functionality as above.

13.2.3 FullyConnected

This target models a fully connected multiprocessor architecture. It forms the base-class for all multiprocessor targets with the fully connected topology. Its parameters are mostly to do with multiprocessor scheduling.

The parameters for FullyConnected are:

nprocs	(`INT`) Default = `2` Number of processors in the target architecture.
sendTime	(`INT`) Default = `1` This is the time required, in processor cycles, to send or receive one datum in the multiprocessor architecture. Sending and receiving are assumed to take the same amount of time.

oneStarOneProc	(`INT`) Default = `NO` If this is `YES`, then all invocations of a star are scheduled onto the same processor.
manualAssignment	(`INT`) Default = `NO` If this is `YES`, then the processor assignment is done manually by the user by setting the `procId` state in each star.
adjustSchedule	(`INT`) Default = `NO` If this is `YES`, then the automatically generated schedule is overridden by manual assignment. This feature requires improvements in the user interface before it can be implemented; hence, the default is `NO`.
childType	(`STRINGARRAY`) Default = `default-CG` This parameter specifies the names of the child targets, separated by spaces. If the number of strings is fewer than the number of processors specified by the `nprocs` parameter, the remaining processors are of type given by the last string. For example, if there are four processors, and *childType* is set to `default-CG56[2] default-CGC`, then the first two child targets will be of type `default-CG56`, and the next two of type `default-CGC`.
resources	(`STRINGARRAY`) Default = The default is the empty string. This parameter defines the specific resources that child targets have, separated by ";". For example, if the first processor has I/O capabilities, this would be specified as `STDIO`. Then, stars that request `STDIO` would be scheduled onto the first processor.
relTimeScales	(`INTARRAY`) Default = `1` This defines the relative time scales of the processors corresponding to child targets. This information is needed by the scheduler in order to compute scheduling costs. The number of entries here should be the same as the number of processors; if not, then the last entry is used for the remaining processors. The entries reflect the relative computing speeds of different processors, and are expressed as relative cycle times. For example, if there is a DSP96000 (32Mhz) and a DSP56000 (20Mhz), the relative cycle times are 1 and 1.6. The default is 1 (meaning that all processors have the same computing speed).
ganttChart	(`INT`) Default = `YES` If this is `YES`, then the Gantt chart containing the generated schedule is displayed.
logFile	(`STRING`) Default = This is the name of the file to which a log will be written of the scheduling process. This is useful for debugging schedulers. If

no file name is specified, no log is generated.

amortizedComm (`INT`) Default = `NO`
If this is `YES`, the scheduler will try to reduce the communication overhead by sending multiple samples per send. This has not really been implemented yet.

schedName(DL,HU,DC,HIER,CGDDF)
(`STRING`) Default = `DL`
Using the *schedName* parameter, a user can select which parallel scheduling algorithm to use. There are three basic SDF parallel scheduling algorithms. The first two can be used for heterogeneous processors, while the last can only be used for homogeneous processors.

`HU` selects a scheduling algorithm based on the classical work by T. C. Hu [Hu61]. This scheduler ignores the interprocessor communication cost (IPC) during scheduling and thus may result in unrealistic schedules. The next two scheduling algorithms take into IPC.

`DL` selects Gil Sih's dynamic level scheduler [Sih93a] (default).

`DC` selects Gil Sih's declustering algorithm [Sih93b]. This scheduler only supports homogeneous multiprocessor targets. It is more expensive than the `DL` and `HU` schedulers, so should be used only if the `DL` and `HU` schedulers produce poor schedules.

`HIER` selects a preliminary version of José Luis Pino's hierarchical scheduler [Pin95]. With this scheduler, the user can specify a top-level parallel scheduler from the three listed above and also specify uniprocessor schedulers for individual galaxies. The default top-level scheduler is `DL`; to specify another use the following syntax: `HIER(HU)` or `HIER(DC)`. To specify a uniprocessor scheduler for a galaxy, add a new galaxy string parameter named *Scheduler* and set it to either `Cluster` (looping level 1), `Loop` (looping level 2) or `SDFScheduler` (looping level 0). See section 13.3.1 for more information on the uniprocessor schedulers.

`CGDDF`[1] selects Soonhoi Ha's dynamic construct scheduler [Ha92]. A dynamic construct, clustered as a star instance, can be assigned to multiple processors. In the future, we may want to schedule a star exploiting data-parallelism. A star instance

1. Note that in Ptolemy0.6, the CGDDF scheduler is not compiled into the default binaries. See "Bugs in pigi" on page A-34 for details.

that can be assigned to multiple processors is called a “macro” actor. `MACRO` scheduler is expected to allow the macro actors. For now, however, `MACRO` scheduler is not implemented.

13.2.4 SharedBus

This third target, also a multiprocessor target, models a shared-bus architecture. In this case, the scheduler computes the cost of the schedule by imposing the constraint that more than one send or receive cannot occur at the same time (since the communication bus is shared).

13.3 Schedulers

Given a Universe of functional blocks to be scheduled and a Target describing the topology and characteristics of the single- or multiple-processor system for which code is to be generated, it is the responsibility of the Scheduler object to perform some or all of the following functions:

- Determine which processor a given invocation of a given Block is executed on (for multiprocessor systems).
- Determine the order in which actors are to be executed on a processor.
- Arrange the execution of actors into standard control structures, like nested loops.

If the program graph follows SDF semantics, all of the above steps are done statically (i.e. at compile time). A dataflow graph with dynamic constructs uses the minimal runtime decision making to determine the execution order of actors.

13.3.1 Single-Processor Schedulers

For targets consisting of a single processor, we provide three different scheduling techniques. The user can select the most appropriate scheduler for a given application by setting the *loopingLevel* target parameter.

In the first approach (*loopingLevel* = DEF), which is the default SDF scheduler, we conceptually construct the acyclic precedence graph (APG) corresponding to the system, and generate a schedule that is consistent with that precedence graph. Note that the precedence graph is not physically constructed. There are many possible schedules for all but the most trivial graphs; the schedule chosen takes resource costs, such as the necessity of flushing registers and the amount of buffering required, into account. The target then generates code by executing the actors in the sequence defined by this schedule. This is a quick and efficient approach when the SDF graph does not have large sample-rate changes. If there are large sample-rate changes, the size of the generated code can be huge because the codeblock for an actor might occur many times (if the number of repetitions for the actor is greater than one); in this case, it is better to use some form of *loop* scheduling.

The second approach we call *Joe’s* scheduling. In this approach (*loopingLevel* = CLUST), actors that have the same sample rate are merged (wherever this will not cause deadlock) and loops are introduced to match the sample rates. The result is a hierarchical clustering; within each cluster, the techniques described above can be used to generate a schedule. The code then contains nested loop constructs together with sequences of code from the actors.

Since the second approach is a heuristic solution, there are cases where some looping possibilities go undetected. By setting the *loopingLevel* to SJS, we can choose the third approach, called *SJS* (Shuvra-Joe-Soonhoi) scheduling after the inventor's first names. After performing Joe's scheduling at the front end, it attacks the remaining graph with an algorithm that is guaranteed to find the maximum amount of looping available in the graph. That is, it generates a single appearance schedule whenever one exists.

A fourth approach, obtained by setting *loopingLevel* to ACYLOOP, we choose a scheduler that generates single appearance schedules optimized for buffer memory usage. This scheduler was developed by Praveen Murthy and Shuvra 'Bhattacharyya [Mur96] [Bha96]. This scheduler only tackles acyclic SDF graphs, and if it finds that the universe is not acyclic, it automatically resets the *loopingLevel* target parameter to SJS. Basically, for a given SDF graph, there could be many different single appearance schedules. These are all optimally compact in terms of schedule length (or program memory in inline code generation). However, they will, in general, require differing amounts of buffering memory; the difference in the buffer memory requirement of an arbitrary single appearance schedule versus a single appearance schedule optimized for buffer memory usage can be dramatic. In code generation, it is essential that the memory consumption be minimal, especially when generating code for embedded DSP processors since these chips have very limited amounts of on-chip memory. Note that acyclic SDF graphs always have single appearance schedules; hence, this scheduler will always give single appearance schedules. If the *file* target parameter is set, then a summary of internal scheduling steps will be written to that file. Essentially, two different heuristics are used by the ACYLOOP scheduler, called APGAN and RPMC, and the better one of the two is selected. The generated file will contain the schedule generated by each algorithm, the resulting buffer memory requirement, and a lower bound on the buffer memory requirement (called BMLB) over all possible single appearance schedules.

If the second, third, or fourth approaches are taken, the code size is drastically reduced when there are large sample rate changes in the application. On the other hand, we sacrifice some efficient buffer management schemes. For example, suppose that star A produces 5 samples to star B which consumes 1 sample at a time. If we take the first approach, we schedule this graph as ABBBBB and assign a buffer of size 5 between star A and B. Since each invocation of star B knows the exact location in the allocated buffer from which to read its sample, each B invocation can read the sample directly from the buffer. If we choose the second or third approach, the scheduling result will be A5(B). Since the body of star B is included inside a loop of factor 5, we have to use indirect addressing for star B to read a sample from the buffer. Therefore, we need an additional buffer pointer for star B (memory overhead), and one more level of memory access (run-time overhead) for indirect addressing.

13.3.2 Multiple-Processor Schedulers

The first step in multiprocessor scheduling, or parallel scheduling, is to translate a given SDF graph to an acyclic precedence expanded graph (APEG). The APEG describes the dependency between invocations of blocks in the SDF graph during execution of one iteration. Refer to the SDF domain documentation for the meaning of one iteration. Hence, a block in a multirate SDF graph may correspond to several APEG nodes. Parallel schedulers schedule the APEG nodes onto processors. Unfortunately, the APEG may have a substantially greater (at times exponential) number of nodes compared to the original SDF graph. For this a hierarchical scheduler is being developed that only partially expands the APEG [Pin95].

We have implemented three basic scheduling techniques that map SDF graphs onto multiple-processors with various interconnection topologies: Hu's level-based list scheduling, Sih's dynamic level scheduling [Sih93a], and Sih's declustering scheduling [Sih93b]. The target architecture is described by its Target object. The `Target` class provides the scheduler with the necessary information on the number of processors, interprocessor communication etc., to enable both scheduling and code synthesis.

The hierarchical scheduler can use any one of the three basic parallel schedulers as the top-level scheduler. The current implementation supports user-specified clustering at galaxy boundaries. These galaxies are assumed to compose into valid SDF stars in which the SDF parameters are derived from the internal schedule of the galaxy. During APEG expansion, these compositions are opaque; thus, the entire galaxy is treated as a single SDF star. Using hierarchical scheduling techniques, we have realized multiple orders of magnitude speedup in scheduling time and multiple orders of magnitude reduction of memory usage. See [Pin95] for more details.

The previous scheduling algorithms could schedule SDF graphs, the `CGDDF` scheduler can also handle graphs with dynamic constructs. See section 13.5 for more details.

Whichever scheduler is used, we schedule communication nodes in the generated code. For example, if we use Hu's level-based list scheduler, we ignore communication overhead when assigning stars to processors. Hence, the generated code is likely to contain more communication code than with the other schedulers that do not ignore the IPC overhead.

There are other target parameters that direct the scheduling procedure. If the parameter *manualAssignment* is set to `YES`, then the default parallel scheduler does not perform star assignment. Instead, it checks the processor assignment of all stars (set using the *procId* state of CG and derived stars). By default, the *procId* state is set to -1, which is an illegal assignment since the child target is numbered from 0. If there is any star, except the `Fork` star, that has an illegal *procId* state, an error is generated saying that manual scheduling has failed. Otherwise, we invoke a list scheduler that determines the order of execution of blocks on each processor based on the manual assignment. We do not support the case where a block might require more than one processor. The *manualAssignment* target parameter automatically sets the *oneStarOneProc* state to `YES`; this is discussed next.

If there are sample rate changes, a star in the program graph may be invoked multiple times in each iteration. These invocations may be assigned to multiple processors by default. We can prevent this by setting the *oneStarOneProc* state to `YES`. Then, all invocations of a star are assigned to the same processor, regardless of whether they are parallelizable or not. The advantage of doing this is the simplicity in code generation since we do not need to splice in `Spread/Collect` stars, which will be discussed later. Also, it provides us another possible scheduling option, *adjustSchedule*; this is described below. The main disadvantage of setting *oneStarOneProc* to `YES` is the performance loss of not exploiting parallelism. It is most severe if Sih's declustering algorithm is used. Therefore, Sih's declustering algorithm is not recommended with this option.

In this paragraph, we describe a future scheduling option that this release does not support yet. Once automatic scheduling (with *oneStarOneProc* option set) is performed, the processor assignment of each star is determined. After examining the assignment, the user may want to override the scheduling decision manually. It can be done by setting the *adjustSchedule* parameter. If that parameter is set, after the automatic scheduling is performed, the *procId* state of each star is automatically updated with the assigned processor. The programmer can override the scheduling decision by changing the value of the *procId* state. The *adjustSchedule* parameter cannot be `YES` before any scheduling decision has been made previously. Again, this option is not supported in this release.

Regardless of which scheduling options are chosen, the final stage of the scheduling is to decide the execution order of stars including send/receive stars. This is done by a simple list scheduling algorithm in each child target. The final scheduling results are displayed on a Gantt chart.

The Gantt Chart Display

Demos that use targets derived from `CGMultiTarget` can produce an interactive Gantt chart display for viewing the parallel schedule.

The Gantt chart display involves a single window for displaying the Gantt chart, which provides scroll bars and zoom buttons for controlling how much of the Gantt chart is shown in the display canvas.

The display canvas represents each star schedule as a box drawn through the time interval over which it is scheduled. If the name of a star can fit in its box, it is printed inside. A vertical bar inside the canvas identifies stars which cannot be labeled. The names of the stars which this bar passes through are printed alongside their respective processor numbers. The bar can be moved horizontally by pressing the left mouse button while on the star to be identified. The stars which the bar passes through are identified by having their icons highlighted in the `vem` window.

Here is a summary of commands that can be used while the Gantt chart display is active:

To change the area of the Gantt chart inside the display canvas:

Use the scroll bars to move along the Gantt chart in the direction desired.

Click on the zoom buttons to increase or decrease the size of the Gantt chart.

To move the vertical bar to the mouse inside the display window:

Depress and drag the left mouse button inside the display window. The left and right cursor keys move the bar by one time interval; shift-left and shift-right move the bar by ten time intervals.

To exit the Gantt chart display program:

Type control-D inside the display window or click on the dismiss button.

The Gantt chart can also be run as a standalone program to display a schedule previously saved by the Gantt chart:

```
gantt schedule_filename
```

A number of limitations exists in the Gantt chart display widget. There are a fixed (hard-coded) number of colors available for coloring processors and highlighting icons. The print function does not work because the font chosen by the font manager is not guaranteed to be Postscript convertible. The save function saves the schedule in the Ptolemy 0.5 format which is different from the Ptolemy 0.6 format generated by the various domains.

13.4 Interfacing Issues

For the 0.6 release, we have developed a framework for interfacing code generation targets with other targets (simulation or code generation). The concepts behind this new infrastructure are detailed in [Pin96]. Currently, only a few of our code generation targets support this new infrastructure including: CGCTarget (CGC Domain), S56XTarget (CG56 Domain), SimVSSTarget (VHDL Domain).

The code generation targets that support this infrastructure can be mixed arbitrarily in an application specification, and can also be embedded within simulation wormholes (i.e. a CG domain galaxy embedded within a simulation-SDF galaxy).

This infrastructure requires that each target provide CGC communication stars that can be targeted to the Ptolemy host workstation. The current implementation does not support specialized communication links between two individual code generation targets, but rather builds the customized links from the communication primitives written in C. To learn how to support a new target in this infrastructure, refer to the *Code Generation* chapter in the *Programmer's Manual*.

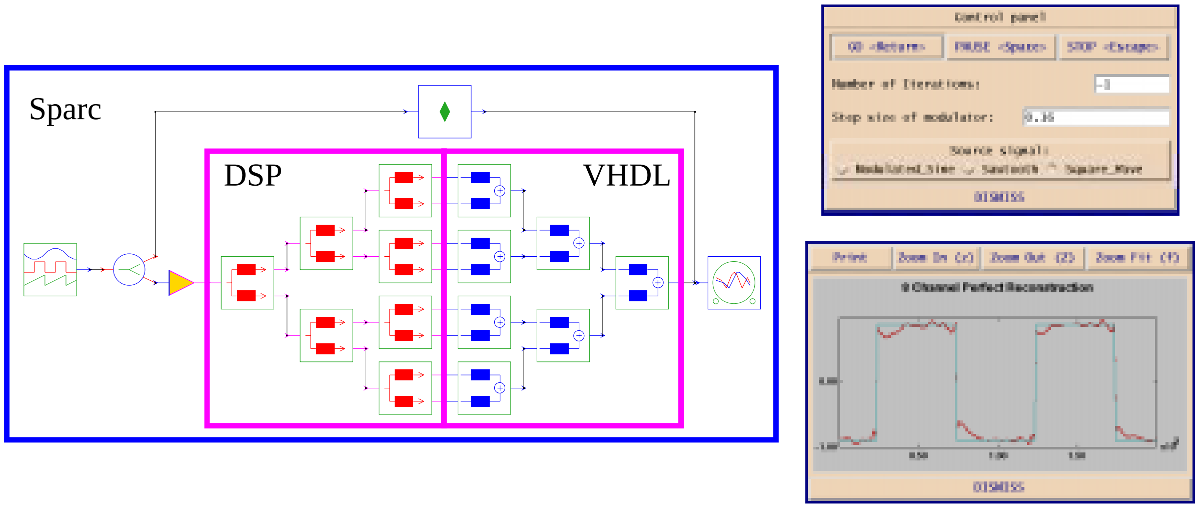
13.4.1 Interface Synthesis between Code Generation Targets

To interface multiple code generation targets, you must set the target parameter for the top-level galaxy to CompileCGSubsystems. The target parameters for CompileCGSubsystems are identical to those of the FullyConnected target, detailed in section 13.2.3. You must declare each individual target in the *childType* CompileCGSubsystems target parameter list. The first of these child targets must be a CGC target whose code will be run on the Ptolemy host workstation. The processor mapping of each star is user-specified by setting either the procId star parameter or setting the domain for the current galaxy. The interconnect between the stars to be mapped onto different targets can be totally arbitrary. A demonstration (included in the release) which mixes targets in the VHDL, CG56 and CGC domains is shown in figure 13-1.

13.4.2 Interface Synthesis between Code Generation and Simulation Domains

The interfacing of code generation targets with simulation targets is more restricted than interfacing only code generation targets. Unlike the previous case, where the star interconnect could be arbitrary, we require that the simulation targets be used at a higher level in the user-specification than all of the code generation targets. This restriction enables us to create simulation SDF star wrappers for each of the code generation subsystems. This generated star can then be added to the user star palette by creating an icon for it using the pigi *make-star* command (See “Editing Icons” on page 2-34.).

The top-level galaxy for each code generation subsystem should have its target set to either CompileCGSubsystems or CreateSDFStar. The CompileCGSubsystems target should be used if more than one code generation target is used. The *childType* target parameter (described in the previous section) should list the child targets to use. The first child target listed must be the CreateSDFStar target. The CreateSDFStar is actually a CGC target that gen-



Number of Iterations: -1

Step size of modulator: 0.16

Source signal:
Modulated_Sine Sawtooth Square_Wave

FIGURE 13-1: Eight channel perfect reconstruction filter bank demonstration using a DSP card, a VHDL simulator and a UNIX workstation. The generated GUI for the application is shown on the right.

erates ptlang code for all of the communication between the various targets and Ptolemy.

If only CGC stars are being used in a code generated subsystem, we have no need for the multiprocessor target CompileCGSubsystems, but rather can use the uniprocessor CGC target CreateSDFStar.

13.5 Dynamic constructs in CG domain

All multiprocessor code generation domains included in previous releases assumed that the dataflow graph is synchronous (or SDF)—that is, the number of tokens consumed and produced by each star does not vary at run time. We also assumed that the relative execution times of blocks was specified, and did not allow blocks with dynamic behavior, such as the *case* construct, data-dependent iteration, and recursion. In simulation, however, data-dependent behavior was supported by the DDF (Dynamic Dataflow) domain. The current release allows data-dependent constructs in the code generation domains, by a new clustering technique and a new scheduler called the CGDDF scheduler[1].

13.5.1 Dynamic constructs as a cluster

Dynamic construct are specified using predefined graph topologies. For example, an *if-then-else construct* is represented as a galaxy that consists of two DDF stars, `Case` and `End-Case`, and two SDF galaxies to represent the bodies of the `TRUE` or `FALSE` branches. The dynamic constructs supported by the CGDDF scheduler are *case*, *for*, *do-while*, and *recursion*. The *case* construct is a generalization of the more familiar *if-then-else* construct. The topology of the galaxy is matched against a set of pre-determined topologies representing these dynamic constructs.

1. In version 0.4 of Ptolemy, dynamic constructs were supported with a separate domain called the CGDDF domain. We have since designed a mechanism for wormhole interfaces to support the CGDDF domain inside the CG domain. By using clustering instead of wormholes, we were able to clean up the code significantly in this release

Galaxy is a hierarchical block for structural representation of the program graph. When an APEG is generated from an SDF graph for parallel scheduling, galaxies are flattened. To handle a dynamic construct as a unit of parallel scheduling, we make a cluster, called a *galaxy cluster*, for each dynamic construct. The programmer should indicate the galaxies to be clustered by creating a galaxy parameter *asFunc* and setting its value to YES. For example, the galaxies associated with the TRUE and the FALSE branch of a *case* construct will have the *asFunc* parameter as well as the galaxy of the construct itself.

13.5.2 Quasi-static scheduling of dynamic constructs

We treat each dynamic construct as a special SDF star and use a static scheduling algorithm. This SDF star is special in the sense that it may need to be mapped onto more than one processor, and the execution time on the assigned processor may vary at runtime (we assume it is fixed when we compute the schedule). The scheduling results decide the assignment to and ordering of blocks on the processors. At run time, we will not achieve the performance expected from the compile time schedule, because the dynamic constructs behave differently to the compile-time assumptions. The goal of the CGDDF scheduler is to minimize the expected makespan of the program graph at run time.

The type of the dynamic construct and the scheduling information related to the dynamic constructs are defined as galaxy parameters. We assume that the run-time behavior of each dynamic construct is known or can be approximated with a certain probability distribution. For example, the number of iterations of a *for* or *do-while* construct is such a variable; similarly, the depth of recursion is a variable of the recursion construct. The parameters to be defined are as follows:

constructType	(`STRING`) Default = There is no default, the initial value is the value of the galaxy parameter. Type of the dynamic construct. Must be one of `case`, `for`, `doWhile`, or `recur` (case insensitive).
paramType	(`STRING`) Default = `geometric` Type of the distribution. Currently, we support `geometric` distribution, `uniform` distribution, and a `general` distribution specified by a table.
paramGeo	(`FLOAT`) Default = `0.5` Geometric constant of a geometric distribution. Its value is effective only if the geometric distribution is selected by *paramType*. If *constructType* is `case`, this parameter indicates the probability of branch 1 (the TRUE branch) being taken. If there are more than two branches, use *paramFile* to specify the probabilities of taking each branch.
paramMin	(`INT`) default = `1` Minimum value of the uniform distribution, effective only when the `uniform` distribution is chosen.
paramMax	(`INT`) default = `10`

	Maximum value of the uniform distribution, effective only when the `uniform` distribution is chosen.
paramFile	(`STRING`) default = `defParams` The name of a file that contains the information on the general distribution. If the construct is a *case* construct, each line contains the probability of taking a branch (numbered from 0). Otherwise, each line contains the integer index value and the probability for that index. The indices should be in increasing order.

Based on the specified run-time behavior distribution, we determine the compile-time profile of each dynamic construct. The profile consists of the number of processors assigned to the construct and the (assumed) execution times of the construct on the assigned processors. Suppose we have a *for* construct. If the loop body is scheduled on one processor, it takes 6 time units. With two processors, the loop body takes 3 and 4 time units respectively. Moreover, each iteration cycle can be paralleled if skewed by 1 time unit. Suppose there are four processors: then, we have to determine how many processors to assign to the construct and how many times the loop body will be scheduled at compile time. Should we assign two processors to the loop body and parallelize two iteration cycles, thus taking all 4 processors? Or should we assign one processor to the loop body and parallelize three iteration cycles, thus taking 3 processors as a whole? The CGDDF scheduler uses a systematic approach based on the distribution to answer these tricky scheduling problems [Ha92]. We can manually determine the number of assigned processors by defining a *fixedNum* galaxy parameter. Note that we still have to decide how to schedule the dynamic construct with the given number of processors. The Gantt chart display will show the profile of the dynamic construct.

13.5.3 DDF-type Stars for dynamic constructs

A code generation domain should have DDF stars to support dynamic constructs with the CGDDF scheduler. For example, the `Case` and `EndCase` stars are used in the *case*, *do-while*, and *recursion* constructs, which differ from each other in the connection topology of these DDF stars and SDF galaxies. Therefore, if the user wants to use one of the above three dynamic constructs, there is no need to write a new DDF star. Like a DDF star, the `Case` star has dynamic output portholes as shown in the `CGCCase.pl` file. For example:

```
 outmulti {
      name { output }
      type { =input }
      num { 0 }
}
```

The *for* construct consists of an *UpSample* type star and a *DownSample* type star, where UpSample and DownSample are not the star names but the types of the stars: if a star produces more than it consumes, it is called an UpSample star. In the preprocessor file, we define a method `readTypeName`, as shown below.

```
method {
      name { readTypeName }
      access { public }
      type { "const char *" }
      code { return "UpSample"; }
}
```

Examples of UpSample type stars are `Repeater` and `DownCounter`. These stars have a data input and a control input. The number of output data tokens is the value of the integer control input, and is thus data-dependent. Conversely, we can design a DownSample star that has the following method:

```
method {
      name { readTypeName }
      access { public }
      type { "const char *" }
      code { return "DownSample"; }
}
```

Examples of DownSample type stars are `LastOfN`, and `SumOfN`. These stars have a data input and a control input. The number of input tokens consumed per invocation is determined by the value of the control input.

As explained above, all customized DDF-type stars for dynamic constructs will be either an UpSample type or a DownSample type. We do not expect that a casual user will need to write new DDF stars if we provide some representative UpSample and DownSample stars in the corresponding code generation domains. Currently, we have DDF stars in the CGC code generation domain only.

13.6 Stars

As mentioned earlier, stars in the CG domain are used only to test and debug schedulers. Thus the stars in the palette shown in figure 13-2 on page 13-15 act generate only comments, and allow the user to model star parameters that are relevant to schedulers such as the number of samples produced and consumed on each firing, and the execution time of the star. By default, any star that is derived from `CGStar` (the base class for all code generation stars), including all the stars in the CG domain, have the state *procId*. This state is used during manual partitioning to specify the processor that the star should be scheduled on. The default value of the state is `-1` which specifies to the scheduler that automatic partitioning should be used. Processors are numbered 0,1,2,...; hence, if the state is set to `1`, then the star will be scheduled on the second processor in the architecture. Note that the target parameter *manualAssignment* should be `YES` for this to work; if *manualAssignment* is `NO`, then the value of *procID* will be ignored (due to a bug in the current implementation). If the user wants to specify a processor assignment for only a subset of the stars in the system, and do automatic assignment for the remaining stars, then this is currently not possible. It can be done in a roundabout manner using the *resources* parameter. This is done by defining a *resources* state in the star. The value of this state is a number that specifies the processor on which this star should go on. The target parameter *resources* is left empty. Then, the scheduler will interpret the value of the *resources* state as the processor on which the star should be scheduled; stars that do not specify any resources are mapped automatically by the scheduler.

The *resources* state just described is used mainly for specifying any special resources that the star might require in the system. For example, an A/D converter star might require an input port, and this port is accessible by only a subset of all the processors in the system; in this case, we would like the A/D star to be scheduled on a processor that has access to the input port. In order to specify this, the *resources* state in the star is defined and set to a string

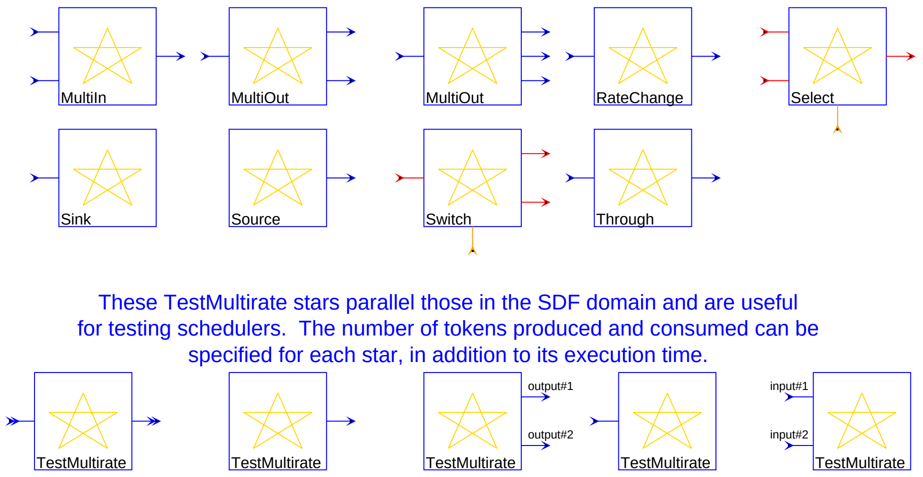

MultiIn
MultiOut
MultiOut
RateChange
Select
Sink
Source
Switch
Through
These TestMultirate stars parallel those in the SDF domain and are useful for testing schedulers. The number of tokens produced and consumed can be specified for each star, in addition to its execution time.
TestMultirate
TestMultirate
output#1
output#2
TestMultirate
TestMultirate
input#1
input#2
TestMultirate

FIGURE 13-2: The CG stars palette.

containing the name of the resource (e.g., `input_port`). Use commas to delimit multiple resources (e.g., `input_port,output_port`). The target parameter *resources* is specified using the same resource names (e.g., `input_port`) as explained in section 13.2.3 on page 13-3. The scheduler will then schedule stars that request certain resources on processors that have them. By default, stars do not have the *resources* state.

The following gives an overview of CG domain stars.

`MultiIn`	Takes multiple inputs and produces one output.
`MultiInOut`	Takes multiple inputs and produces multiple outputs.
`MultiOut`	Takes one input and produces multiple outputs.
`RateChange`	Consumes *consume* samples and produces *produce* samples.
`Sink`	Swallows an input sample.
`Source`	Generic code generator source star; produces a sample.
`Switch`	This star requires a BDF scheduler. It switches input events to one of two outputs, depending on the value of the control input.
`Through`	Passes data through. The run time can be set to reflect computation time.
`TestMultirate`	(five icons) The `TestMultirate` stars parallel those in the SDF domain. These stars are useful for testing schedulers. The number of tokens produced and consumed can be specified for each star, in addition to its execution time.

13.7 Demos

There are four demos in the CG domain, shown in figure 13-3; these are explained below.

FIGURE 13-3: Code Generation demonstrations

`pipeline`	This demo demonstrates a technique for generation of pipelined schedules with Ptolemy's parallel schedulers, even though Ptolemy's parallel schedulers attempt to minimize *makespan* (the time to compute one iteration of the schedule) rather than maximize the throughput (the time for each iteration in the execution of a very large number of iterations). To *retime* a graph, we simply add delays on all feedforward arcs (arcs that are not part of feedback loops). We must not add delays in feedback loops as that will change the semantics. The effect of the added delays is to cause the generation of a pipelined schedule. The delays marked as "(conditional)" in the demo are parameterized delays; the delay value is zero if the universe parameter *retime* is set to `NO`, and is 100 if the universe parameter is set to `YES`. The delay in the feedback loop is always one. Schedules are generated in either case for a three-processor system with no communication costs. If this were a real-life example, the programmer would next attempt to reduce the "100" values to the minimum values that enable the retimed schedule to run; there are other constraints that apply as well when there are parallel paths, so that corresponding tokens arrive at the same star. If the system will function correctly with zero values for initial values at points where the retiming delays are added, the generated schedule can be used directly. Otherwise, a *preamble*, or partial schedule, can be prepended to provide initial values.
`schedTest`	This is a simple multiprocessor code generation demo. By changing the parameters in the RateChange star, you can make the demo more interesting by observing how the scheduler manages to parallelize multiple invocations of a star.
`Sih-4-1`	This demo allows the properties of the parallel scheduler to be investigated, by providing a universe in which the run times of stars, the number of processors, and the communication cost between processors can be varied. The problem, as presented by the default parameters, is to schedule a collection of dataflow actors on three processors with a shared bus connecting them. Executing the demo causes a Gantt chart display to appear, showing the partitioning of the actors onto the three processors. Clicking the left mouse button at various points in the schedule

causes the associated stars to be highlighted in the universe palette. After exiting from the Gantt chart display, code is written to a separate file for each processor (here the “code” is simply a sequence of comments written by the dummy CG stars). It is interesting to explore the effects of varying the communication costs, the number of processors, and the communication topology. To do so, execute the *edit-target* command (type 'T'). A display of possible targets comes up. Of the available options, only `SharedBus` and `FullyConnected` will use the parallel scheduler, so select one of them and click on "Ok". Next, a display of target parameters will appear. The interesting ones to vary are *nprocs*, the number of processors, and *sendTime*, the communication cost. Try using two or four processors, for example. Sometimes you will find that the scheduler will not use all the processors. For example, if you make the communication cost very large, everything will be placed on one processor. If the communication cost is 1 (the default), and four processors are provided, only three will be used.

`useless` This is a simple demo of the dummy stars provided in the CG domain. Each star, when executed, adds code to the target. On completion of execution for two iterations, the accumulated code is displayed in a popup window, showing the sequence of code produced by the three stars.

Chapter 14. CGC Domain

Authors: *Joseph Buck*
Soonhoi Ha
Christopher Hylands
Edward A. Lee
Thomas M. Parks

Other Contributors: *Kennard White*
Mary Stewart

14.1 Introduction

The CGC domain generates code for the C programming language. This domain supports both synchronous dataflow (SDF, see “SDF Domain” on page 5-1) and Boolean-controlled dataflow (BDF, see “BDF Domain” on page 8-1) models of computation. The model associated with a particular program graph is determined by which target is selected. The `bdf-CGC` target supports the BDF model, while all other targets in the CGC domain support only the SDF model. Code can be generated for both single-processor and multi-processor computers. The targets that support single processors include `default-CGC`, `Makefile_C`, `TclTk_Target`, and `bdf-CGC`. The multi-processor targets are `unixMulti_C` and `NOWam`.

14.2 CGC Targets

The targets of the CGC domain generate C code from dataflow program graphs. Code generation is controlled by the *host*, *directory*, and *file* parameters as described in “Targets” on page 13-1. The command used to compile the code is determined by the *compileCommand*, *compileOptions*, and *linkOptions* parameters. Compilation and execution are controlled by the *display?*, *compile*?, and *run?* parameters, also described in “Targets” on page 13-1. The other parameters common to all CGC targets are listed below. Not all of these parameters are made available to the user by every target, and some targets define additional parameters.

staticBuffering	(`INT`) Default = `TRUE` If `TRUE`, then attempt to use static, compile-time addressing of data buffers between stars. Otherwise, generate code for dynamic, run-time addressing.
funcName	(`STRING`) Default = `main` The name of the main function. The default value of `main` is suitable for generating stand-alone programs. Choose another name if you wish to use the generated code as a procedure that is called from your own main program.
compileCommand	(`STRING`) Default = `cc` Command name of the compiler.

compileOption	(`STRING`) Default = Options passed to the compiler. The default is the empty string.
linkOptions	(`STRING`) Default = `-lm` Options passed to the linker.
resources	(`STRING`) Default = `STDIO` List of abstract resources that the host computer has.

14.2.1 Single-Processor Targets

The `default-CGC` target generates `C` code for a single processor from a SDF program graph. The parameters available to the user are shown in Table 14-1, "Parameters of the default-CGC target," on page 14-2. See "Targets" on page 13-1 and "CGC Targets" on page 14-1 for detailed descriptions of these parameters.

`compile?`	`file`	`Looping Level`
`compileCommand`	`funcName`	`resources`
`compileOptions`	`host`	`run?`
`directory`	`linkOptions`	`staticBuffering`
`display?`		

TABLE 14-1: Parameters of the `default-CGC` target

The `Makefile_C` target compiles CGC binaries with makefiles so that compile time architecture and site dependencies can be handled. The `Makefile_C` target generates a small makefile that is `rcp`'d over to the remote machine. The generated makefile is named after the universe. If the universe is called `bigBang`, then the makefile will be called `bigBang.mk`. We name the generated makefiles so that more than one makefile can exist in the users' directory.

The generated makefile uses `$PTOLEMY/lib/cgc/makefile_C.mk` as a starting point, and then appends lines to it. The generated makefile includes `$PTOLEMY/mk/config-$PTARCH.mk`, which determines architecture and site dependencies, such as which compiler to use, or where the X11 include files are. The user may modify `makefile_C.mk` and add site-dependent rules and variables there. If the user wants to have site dependent include files on the remote machines, then they could add `include $(ROOT)/mk/mysite.mk` to `makefile_C.mk`, and that file would be included on the remote machines at compile time.

On the remote machine, the `Makefile_C` target assumes:

- `$PTOLEMY` and `$PTARCH` are set on the remote machine when `rsh`ing.
- `$PTOLEMY/mk/config-$PTARCH.mk` and any makefile files included by that file are present.
- A `make` binary is present. The `Makefile_C` target does not assume GNU `make`, so the default `makefile_C.mk` does not include `mk/common.mk`. The reason not to assume GNU `make` is that we are not sure what the user's path is like when they log in. The user can require that GNU `make` be used by setting the *skeletonMakefile* target parameter to the name of a makefile that requires GNU `make`.

If the remote machine does not fulfill these constraints, then the user should use the

`Default_C` target.

skeletonMakefile	(`STRING`) Default= The default value of this target parameter is the empty string, which means that we use `$PTOLEMY/lib/cgc/makefile_C.mk` as our skeleton makefile. If this parameter is not empty then the value of the parameter refers to the skeleton makefile to be copied into the generated makefile.
appendToMakefile	(`INT`) Default = `1` This target parameter controls whether we append rules to the generated makefile or just copy it over to the remote machine. In the default situation, *appendToMakefile* is true and we append our rules after copying `$PTOLEMY/lib/cgc/makefile_C.mk`

The parent target of the `Makefile_C` target is `default-CGC`. If the parent target parameter *compileOptions* is set, then we process any environment variables in that string, and then add it to the end of the generated makefile as part of `OTHERCFLAGS=`. In a similar fashion, the parent target parameter *linkOptions* ends up as part of the right-hand side of `LOADLIBES=`.

The `TclTk_Target` target, which is derived from the `Makefile_C` target, must be used when Tcl/Tk stars are present in the program graph. The initial default of one parameter differs from that of the parent target.

skeletonMakefile	(`STRING`) Default=`$PTOLEMY/lib/cgc/TclTk_Target.mk` The TclTk_Target overrides this parent target parameter and sets it to the name of a skeleton makefile to be copied into the generated makefile.

The `bdf-CGC` target supports the BDF model of computation. It must be used when BDF stars are present in the program graph. It can also be used with program graphs that contain only SDF stars. The `bdf-CGC` target has the same parameters as the `default-CGC` target with the exception that the *Looping Level* parameter is absent. This is because a loop-generating algorithm is always used for scheduling. See “BDF Domain” on page 8-1 for details.

14.2.2 Multi-Processor Targets

Currently, the CGC domain supports two multi-processor targets: `unixMulti_C` and `NOWam`. The `unixMulti_C` target generates code for multiple networked workstations using a shared bus configuration for scheduling purposes. Inter-processor communication is implemented by splicing send/receive stars into the program graph. These communication stars use the TCP/IP protocol. In addition to the target parameters described in “CGC Targets” on page 14-1 and “Targets” on page 13-1, this target defines the user parameters listed below. Table 14-2, “Parameters of the unixMulti_C target,” on page 14-4 gives the complete list of parameters for the `unixMulti_C` target.

`adjustSchedule`	`ignoreIPC`	`overlapComm`
`amortizedComm`	`inheritProcessors`	`portNumber`
`childType`	`logFile`	`relTimeScales`
`compile?`	`machineNames`	`resources`
`directory`	`manualAssignment`	`run?`
`display?`	`nameSuffix`	`sendTime`
`file`	`nprocs`	`userCluster`
`ganttChart`	`oneStarOneProc`	`tabular`

TABLE 14-2: Parameters of the `unixMulti_C` target

portNumber	(`INT`) Default = `7654` The starting TCP/IP port number used by send/receive stars. The port number is incremented for each send/receive pair. It is the responsibility of the user to ensure that the port number does not conflict with any that may already be in use.
machineNames	(`STRING`) Default = `herschel` The host names of the workstations which form the multi-processor. The names should be separated by a comma (‘`,`’).
nameSuffix	(`STRING`) Default = The default is the empty string. The domain suffix for the workstations named in *machineNames*. If left blank, which is the default, then the workstations are assumed to be part of the local domain. Otherwise, specify the proper domain name, including a leading period. This string is appended to the names in *machineNames* to form the fully qualified host names.

The `NOWam` target uses Networks Of Workstations (NOW) active messages to communicate between machines. The NOW project is an effort to use many commodity workstations to create a building-wide supercomputer. For more information about the NOW project, see `http://now.cs.berkeley.edu`. Currently, the `NOWam` target is still experimental, and only a proof of concept. The `NOWam` target has the following target parameters:

machineName	(`STRING`) Default = `lucky, babbage` The host names of the workstations which form the multi-processor. The names should be separated by a comma (‘`,`’). The `NOWam` target will not work on the local machines, the machines named by this parameter must be remote machines. Note that the default of this parameter differs from the default in the `UnixMulti_C` target.
nameSuffix	(`STRING`) Default = The default is the empty string. See the description of *nameSuffix* in `UnixMulti_C` above.

14.2.3 Setting Parameters Using Command-line Arguments

The pragma facility allows users to identify any parameters that the user would like to be able to change on the command line of the CGC binary. CGC command line arguments has not been extensively tested yet. Currently, it is only supported for scalar parameters with `FLOAT` and `INT` values. Also, it is only working for parameters of Stars at the top level, i.e. it will not work with Galaxies' and Universes' parameters, or parameters of Stars in Galaxies.

To specify a parameter for setting via the command-line, place the cursor over the Star and invoke the *edit-pragmas* command ('`a`'). In the dialog box, enter the name of the parameter to be made settable, follow by white space, then the name of the command-line option with which to set the parameter. This parameter /option-name pair should be entered for each of the required parameters, with pairs separated by white space.

Now, the generated program will take the new options each followed by a value with which to set the corresponding parameters. If the command-line option is not specified for a parameter, it will be initialized to its default value, which will be the value set by the *edit-params* command ('`e`'). In addition, if the '`-h`', '`-help`' or '`-HELP`' option is specified, the program will print the option-names corresponding to the settable parameters with their default values.

14.3 An Overview of CGC Stars

Figure 14-1 shows the top-level palette of CGC stars. The stars are divided into categories: sources, sinks, arithmetic functions, nonlinear functions, control, Sun UltraSparc VIS-conversion, signal processing, boolean-controlled dataflow, Tcl/Tk and higher-order function (HOF) stars. Icons for `delay`, `bus`, and `BlackHole` appear in most palettes for easy access. Many of the stars in the CGC domain have equivalent counterparts in the SDF domain. See "An overview of SDF stars" on page 5-4 for brief descriptions of these stars. Brief descriptions of the stars unique to the CGC domain are given in the following sections.

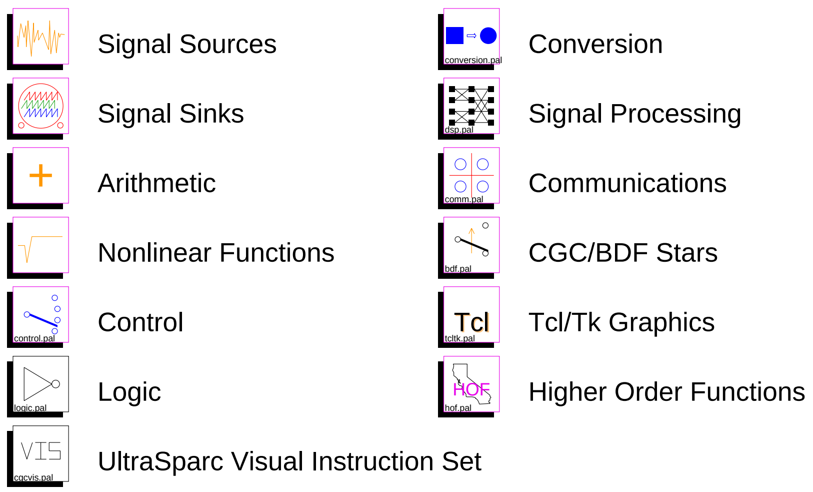


FIGURE 14-1: Top-level palette of stars in the CGC domain

14.3.1 Source Stars

Source stars have no inputs and produce data on their outputs. Figure 14-2 shows the palette of CGC source stars. The following stars are equivalent to the SDF stars of the same name (see “Source stars” on page 5-5): `Const`, `IIDUniform`, `Ramp`, `Rect`, `singen`, `WaveForm`, `TclScript`, `TkSlider`, `RampFix`, `RectFix`, `RampInt`, `expgen`. Stars that are unique to the CGC domain are described briefly below.

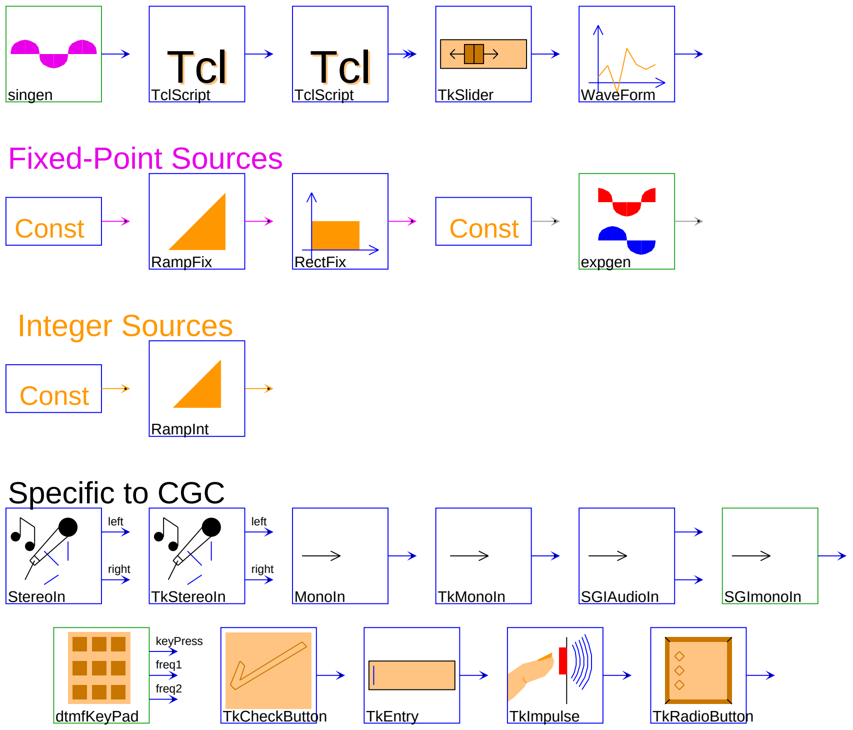


FIGURE 14-2: Source stars in the CGC domain

`StereoIn`	Reads Compact Disc format audio data from a file given by `fileName`. The file can be the audio port `/dev/audio`, if supported by the workstation. The data read is linear 16 bit encoded and stereo (2 channel) format.
`TkStereoIn`	Just like `StereoIn`, except that a Tk slider is put in the master control panel to control the volume.
`MonoIn`	`Reads` mono (1 channel) data with either linear16 or ulaw8 encoding from a file given by `fileName`. The file can be the audio port `/dev/audio` , if supported by the workstation.
`TkMonoIn`	Just like `MonoIn`, except that a Tk slider is put in the master control panel to control the volume.
`SGImonoIn`	(SGI only) Average the stereo audio output of an `SGIAudioIn` star into one mono output.
`SGIAudioIn`	(SGI only) Get samples from the audio input port on an Silicon Graphics workstation.
`dtmfKeyPad`	Generate a Dual-Tone Modulated Frequency (DTMF) signal.

`TkCheckButton`	A simple Tk on/off input source.
`TkEntry`	Output a constant signal with value determined by a Tk entry box (default 0.0).
`TkImpulse`	Output a specified value when a button is pushed. Optionally synchronize by halting until the button is pushed.
`TkRadioButton`	Graphical one-of-many input source.

14.3.2 Sink Stars

Sink stars have no outputs and consume data on their inputs. Figure 14-3 shows the palette of CGC sink stars. The following stars are equivalent to the SDF stars of the same name (see “Sink stars” on page 5-9): `XMgraph`, `XYgraph`, `Xscope`, `TkBarGraph`, `TkPlot`, `TKXYPlot`, `TclScript`, `Printer`. Stars that are unique to the CGC domain are described briefly below.

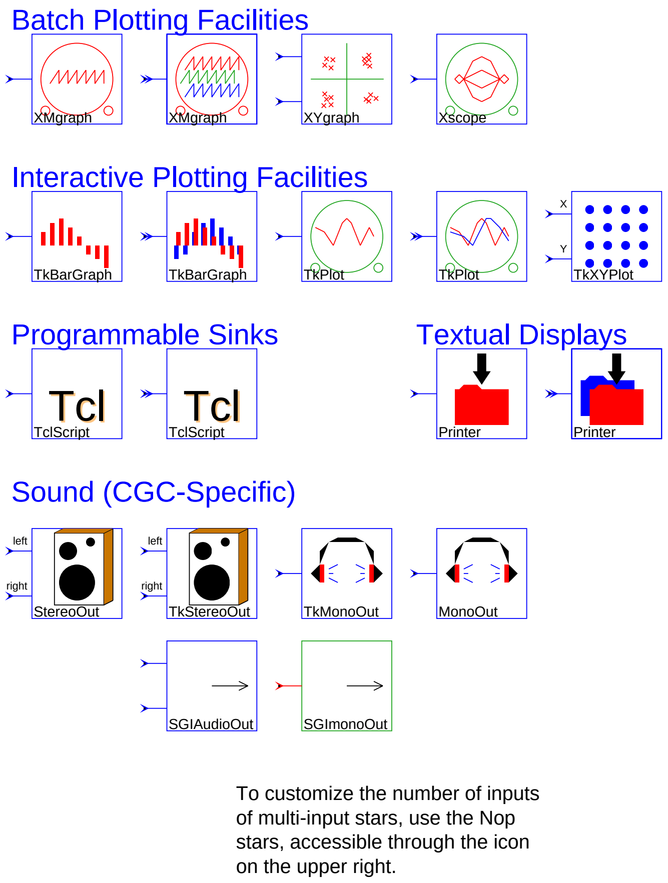


FIGURE 14-3: Sink stars in the CGC domain

`StereoOut`	Writes Compact Disc audio format to a file given by `fileName`. The file can be the audio port `/dev/audio`, if supported by the workstation. The data written is linear 16 bit encoded and stereo (2 channel) format.

`TkStereoOut`	Just like `StereoOut` except that Tk sliders are put in the master control panel to control the volume and balance.
`TkMonoOut`	Just like `MonoOut` except that Tk sliders are put in the master control panel to control the volume.
`MonoOut`	Writes mono (1 channel) data with either linear16 or ulaw8 encoding to a file given by `fileName`. The file can be the audio port `/dev/audio`, if supported by the workstation. If the *aheadlimit* parameter is non-negative, then it specifies the maximum number of samples that the program is allowed to compute ahead of real time.
`SGIAudioOut`	(SGI Only) Put samples into an audio output port.
`SGIMonoOut`	(SGI Only) A galaxy that takes a mono output and drives the stereo `SGIAudioOut` star below.

14.3.3 Arithmetic Stars

Arithmetic stars perform simple functions such as addition and multiplication. Figure 14-4 shows the palette of CGC arithmetic stars. All of the stars are equivalent to the SDF stars of the same name (see “Arithmetic stars” on page 5-12): `Add`, `Gain`, `Integrator`, `Mpy`, `Sub`.

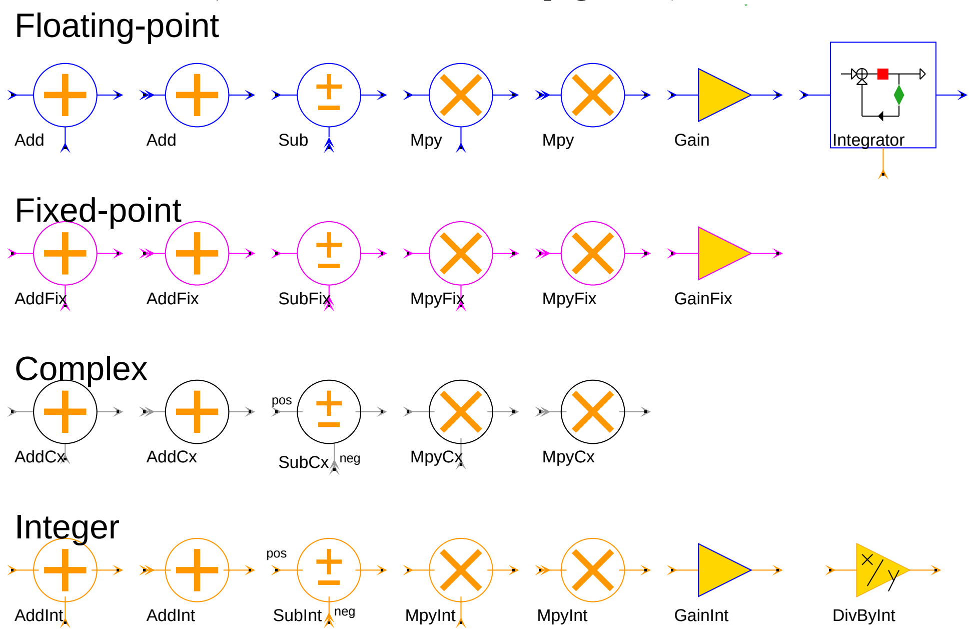


FIGURE 14-4: Arithmetic stars in the CGC domain

14.3.4 Nonlinear Stars

Nonlinear stars perform simple functions. Figure 14-5 shows the palette of CGC nonlinear stars. The following stars are equivalent to the SDF stars of the same name (see “Nonlinear stars” on page 5-13): `Abs`, `cexp`, `conj`, `Cos`, `Dirichlet`, `Exp`, `expjx`, `Floor`, `Limit`, `Log`, `MaxMin`, `Modulo`, `ModuloInt`, `OrderTwoInt`, `Reciprocal`, `Sgn`, `Sin`, `Sinc`, `Sqrt`, `powerEst`, `Quant`, `Table`, `TclScript`. Stars that are unique to the CGC domain are described briefly below.

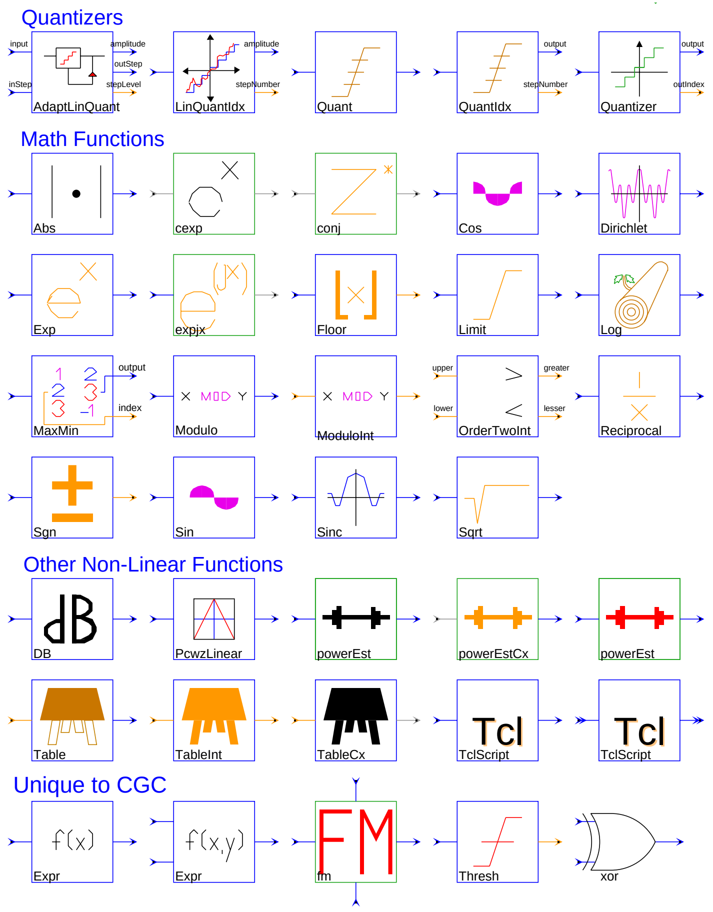


FIGURE 14-5: Nonlinear stars in the CGC domain

`Expr`	(Two icons) General expression evaluation. This star evaluates the expression given by the *expr* parameter and writes the result on the output. The default expression, which is `$ref(in#1)`, simply copies the first input to the output.
`fm`	Modulate a signal by frequency.
`Thresh`	Compares input values to threshold. The output is `0` if input <= threshold, otherwise it is `1`.
`xor`	Exclusive-OR two signals.

14.3.5 Control Stars

Control stars are used for routing data and other control functions. Figure 14-6 shows the palette of CGC control stars. The following stars are equivalent to the SDF stars of the same name (see “Conversion stars” on page 5-20): `Fork`, `Chop`, `ChopVarOffset`, `Commutator`, `DeMux`, `Distributor`, `DownSample`, `Mux`, `Repeat`, `UpSample`. Stars that are unique to the CGC domain are described briefly belowp.

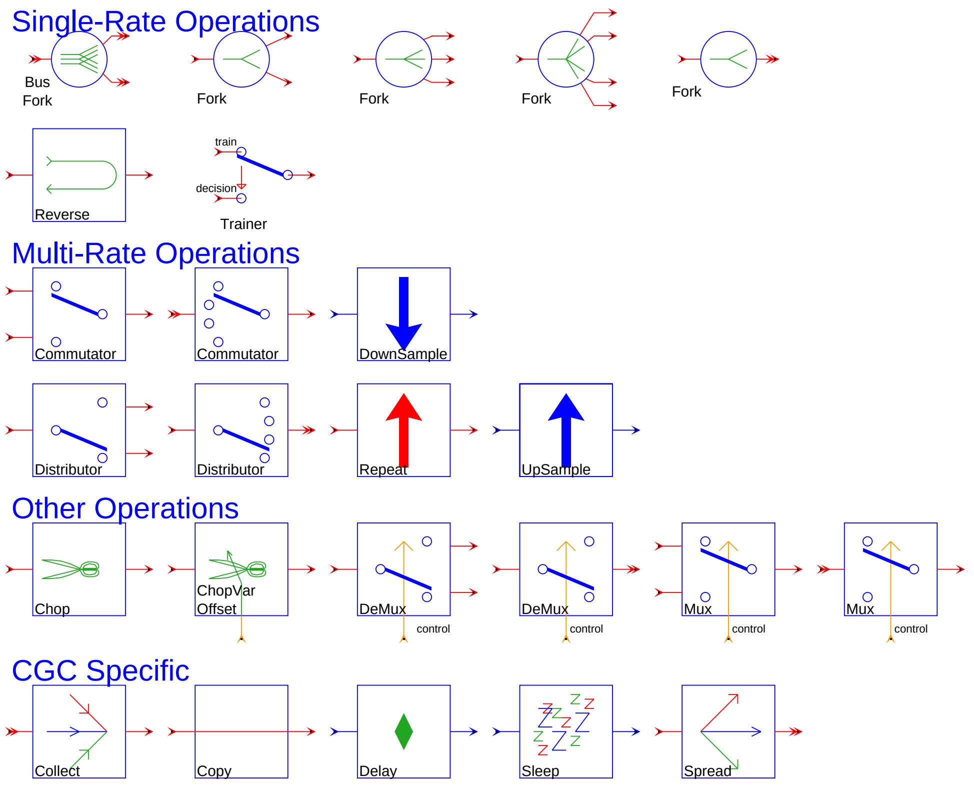


FIGURE 14-6: Control stars in the CGC domain

`Collect`	Takes multiple inputs and produces one output. This star does not generate code. In multiprocessor code generation, it is automatically attached to a porthole if it has multiple sources. Its role is just opposite to that of the `Spread` star.
`Copy`	‘Copy’ stars are added if an input/output PortHole is a host/embedded PortHole and the buffer size is greater than the number of Particles transferred.
`Delay`	Delay an input by *delay* samples.
`Sleep`	Suspend execution for an interval (in milliseconds). The input is passed to the output when the process resumes.
`Spread`	Takes one input and produces multiple outputs. This star does not generate any code. In multiprocessor code generation, this star is automatically attached to a porthole whose outputs are passed to more than one destination (one ordinary block and one `Send` star, more than one `Send` star, and so on.)

14.3.6 Logic Stars

Figure 14-7 shows the palette of CGC Logic stars.

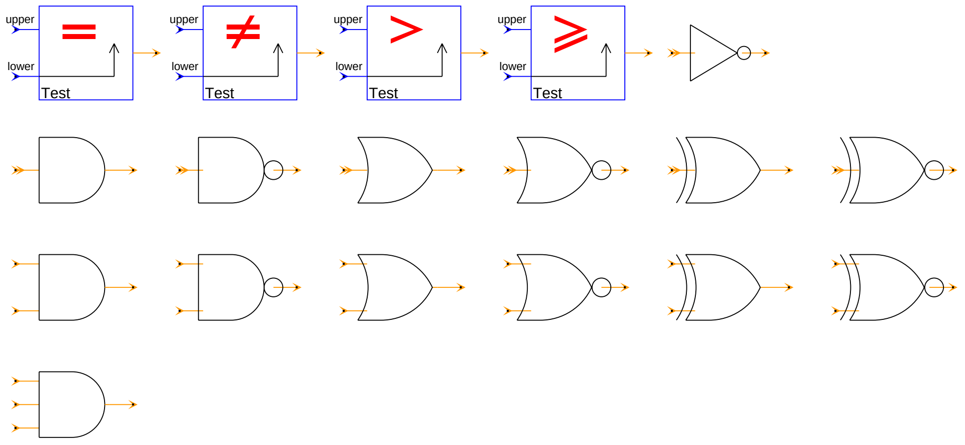


FIGURE 14-7: Logic stars in the CGC domain

14.3.7 Conversion Stars

Conversion stars are used to convert between complex and real numbers. Figure 14-8 shows the palette of CGC conversion stars. All of the stars are equivalent to the SDF stars of the same name (see “Conversion stars” on page 5-20): `CxToRect`, `PolarToRect`, `RectToCx`, `RectToPolar`.

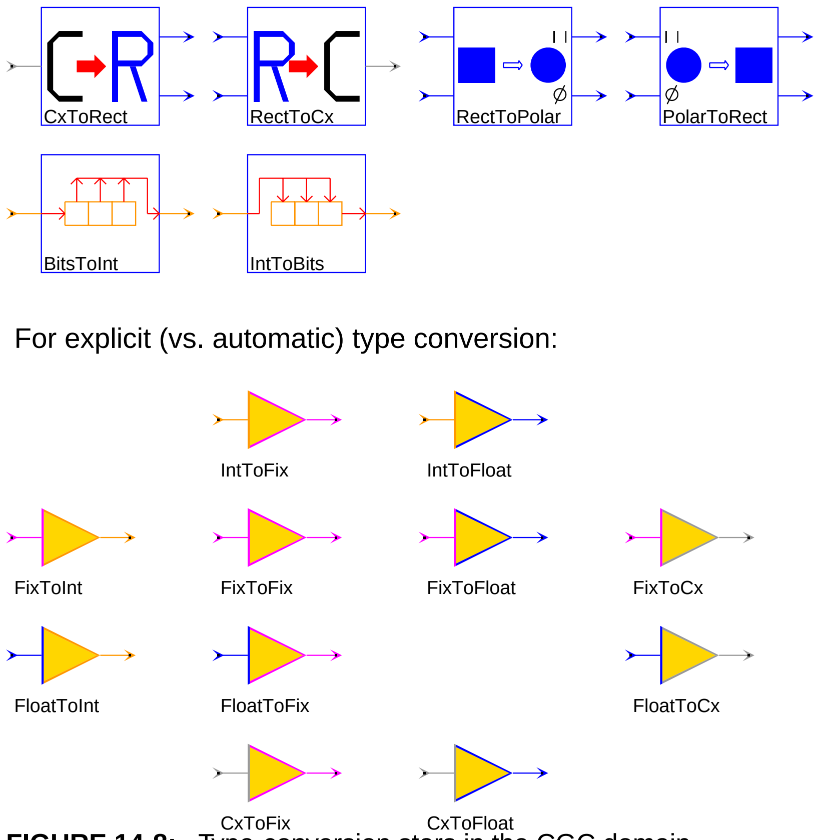


FIGURE 14-8: Type-conversion stars in the CGC domain

14.3.8 Signal Processing Stars

Figure 14-9 shows the palette of CGC signal processing stars. The following stars are equivalent to the SDF stars of the same name (see “Signal processing stars” on page 5-30): `DB`, `FIR`, `FIRFix`, `FFTCx`, `GAL`, `GGAL`, `Goertzel`, `LMS`, `LMSOscDet`, `LMSTkPlot`. The `IIR`, `RaisedCosine` and `Window` CGC stars are not present in Ptolemy0.6. Stars that are unique to the CGC domain are described briefly below.

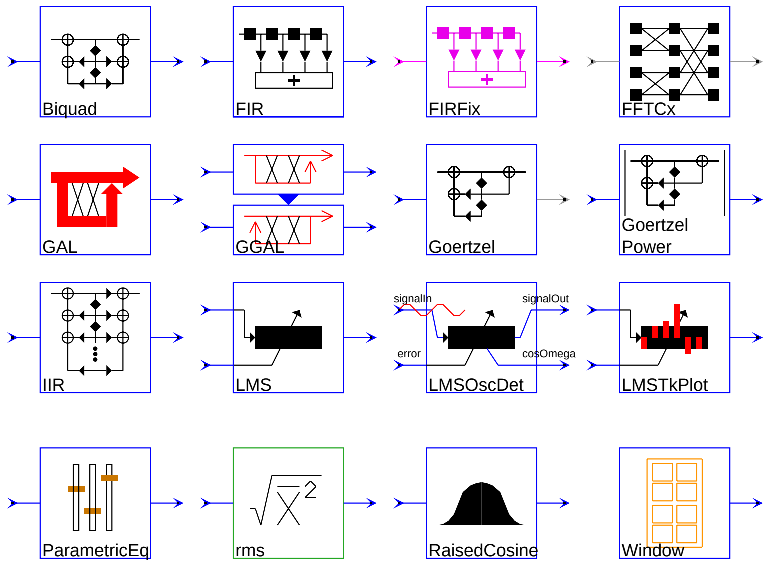


FIGURE 14-9: Signal processing stars in the CGC domain

`GoertzelPower`	Second-order recursive computation of the power of the kth coefficient of an N-point DFT using Goertzel’s algorithm. This form is used in touchtone decoding.
`ParametricEq`	A two-pole, two-zero parametric digital IIR filter (a biquad).
`rms`	Calculate the Root Mean Squared of a signal.

14.3.9 Communications Stars

Figure 14-10 shows the communications stars in the CGC domain. The following stars

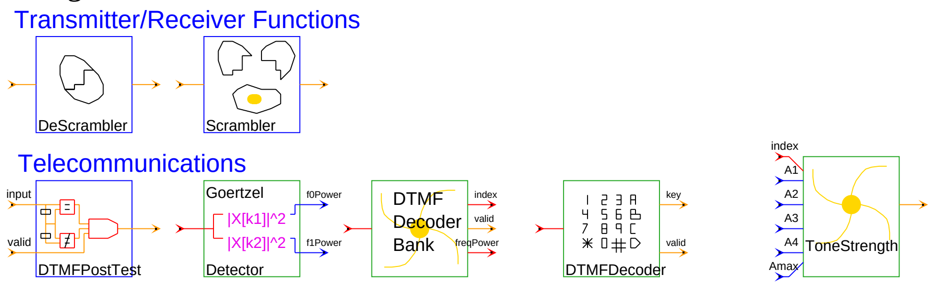


FIGURE 14-10: Communications stars in the CGC domain

are equivalent to the SDF stars of the same name in the communications palette, (See “Communication stars” on page 5-36): `DeScrambler`, `Scrambler`. The following stars are equivalent to the SDF stars of the same name in the telecommunications palette, (see “Telecommunications” on page 5-39): `DTMFPostTest`, `GoertzelDetector`, `DTMFDe-`

`coderBand`, `DTMFDecoder`, `ToneStrength`.

14.3.10 BDF Stars

BDF stars are used for conditionally routing data. Figure 14-11 shows the palette of BDF stars in the CGC domain. These stars require the use of the `bdf-CGC` target (see “Single-Processor Targets” on page 14-2). Unlike their simulation counterparts (see “An overview of BDF stars” on page 8-2), these stars can only transfer single tokens in one firing.

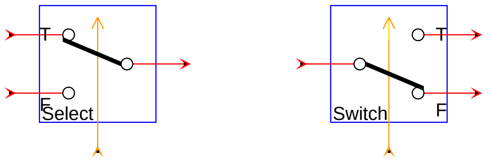


FIGURE 14-11: BDF stars in the CGC domain

`Select`	This star requires a BDF scheduler. If the value on the *control* line is nonzero, *trueInput* is copied to the output; otherwise, *falseInput* is.
`Switch`	This star requires a BDF scheduler. Switches *input* events to one of two outputs, depending on the value of the *control* input. If *control* is true, the value is written to *trueOutput*; otherwise it is written to *falseOutput*.

14.3.11 Tcl/Tk Stars

Tcl/Tk stars require the use of the `TclTk_Target` target. They can be used to provide an interactive user interface with Tk widgets. Figure 14-12 shows the palette of Tcl/Tk stars available in the CGC domain. Most of these stars are described in the sources, sinks and nonlinear palettes.

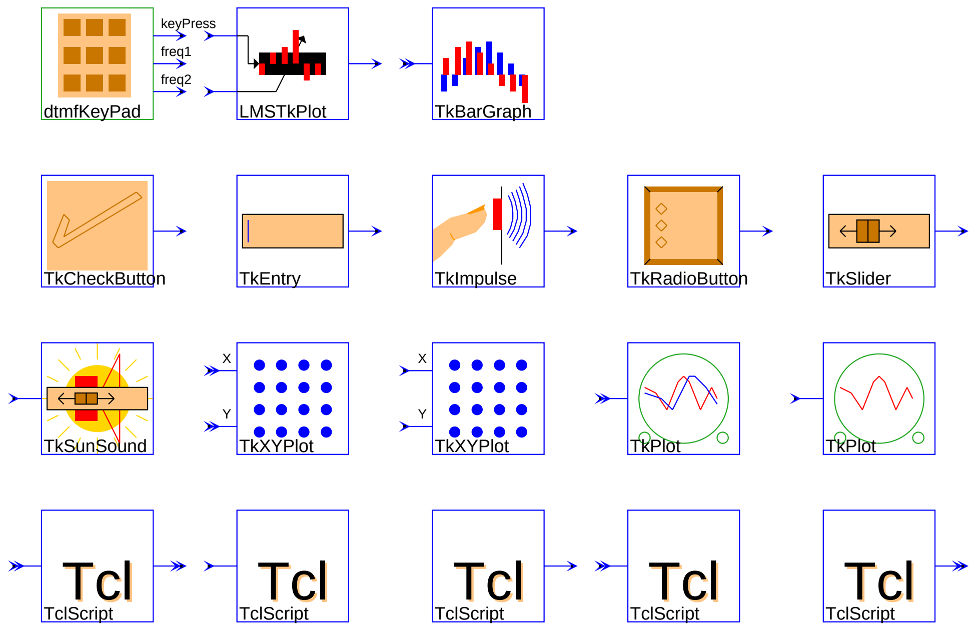


FIGURE 14-12: Tcl/Tk stars in the CGC domain

`TkParametricEq`	Just like `ParametricEq` star, except that a Tk slider is put in the master control panel to control the gain, bandwidth, and center and cut-off frequencies.

14.3.12 Higher Order Function Stars

For information on the HOF stars, please see “An overview of the HOF stars” on page 6-15.

14.3.13 UltraSparc VIS (Visual Instruction Set) Stars

These stars generate code that includes instructions for the UltraSparc’s Visual Instruction Set (VIS). These stars only run on Sun UltraSparc workstations (see “UltraSparc VIS Demos” on page 14-26 for more information about using CGCVIS demos.) All of these stars process data in “quad-words”—64-bit words, each containing four, 16-bit signed integers. All of the stars exhibit some speed improvement over the equivalent stars written in floating-point, although a substantial effort is needed in coding them to realize this performance gain.

CGC VIS universes that create standalone applications, such as the 256fft demo, should use the CGC `Makefile_C` target and set the *skeletonMakefile* target parameter to `$PTOLEMY/lib/cgc/makefile_VIS.mk`. Universes that use CGC VIS stars and TclTk stars should use `makefile_TclTk_VIS.mk`. The `CGCVISSim` target can be used to simulate

VIS stars, but this target is very experimental.

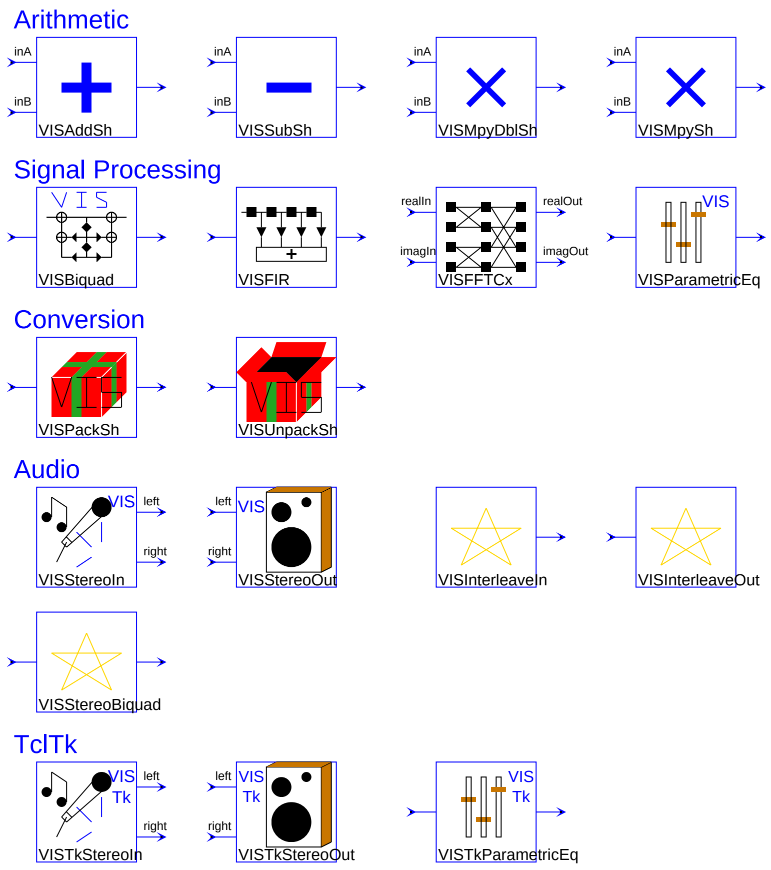


FIGURE 14-13: UltraSparc Visual Instruction Set

`VISAddSh`	Add the corresponding 16-bit fixed point numbers of two partitioned float particles. Four signed 16-bit fixed point numbers of a partitioned 64-bit float particle are added to those of another 64-bit float particle. The result is returned as a single 64-bit float particle. There is no saturation arithmetic so that overflow results in wrap around.
`VISSubSh`	Subtract the corresponding 16-bit fixed point numbers of two-partitioned float particles. Four signed 16-bit fixed point numbers of a partitioned 64-bit float particle are subtracted from those of another 64-bit float particle. The result is returned as a single 64-bit float particle. There is no saturation arithmetic so that overflow results in wrap around.
`VISMpyDblSh`	Multiply the corresponding 16-bit fixed point numbers of two-partitioned float particles. Four signed 16-bit fixed point numbers of a partitioned 64-bit float particle are multiplied to those of another 64-bit float particle. Each multiplication produces a 32-bit result. Each 32-bit result is then left-shifted to fit within

a certain dynamic range and truncated to 16 bits. The final result is four 16-bit fixed point numbers that are returned as a single float particle.

`VISMpySh` Multiply the corresponding 16-bit fixed point numbers of two-partitioned float particles. Four signed 16-bit fixed pointnumbers of a partitioned 64-bit float particle are multiplied to those of another 64-bit float particle. Each multiplication produces a 32-bit result, which is then truncated to 16 bits. The final result is four 16-bit fixed point numbers that are returned as a single float particle.

`VISBiquad` An IIR biquad filter. In order to take advantage of the 16-bit partitioned multiplies, the VIS biquad reformulates the filtering operation to that of a matrix operation (Ax=y), whereVIS A is a matrix calculated from the taps, x is an input vector, and y is an output vector.The matrix A is first calculated by substituting the biquad equation y[n] = -a*y[n-1]-b*y[n-2]+c*x[n]+d*x[n-1]+e*x[n-2] into y[n-1], y[n-2], and y[n-3]. The matrix A is then multiplied with the 16-bit partitioned input vector. The final result is accumulated in four 16-bit fixed point numbers which are concatenated into a single 64-bit float particle.

`VISFIR` A finite impulse response (FIR) filter. In order to take advantage of the 16-bit partitioned multiplies, the VIS FIR reformulates the filtering operation to that of a matrix operation (Ax=y), where A is a tap matrix, x is an input vector, and y is an outputvector. The matrix A is first constructed from the filter taps. Each row is filled by copying the filter taps, zero-padding so that its length is a multiple of 4, and shifting to the right by one. Four of these rows are used to build up matrix A. The matrix A is then multiplied with the 16-bit partitioned input vector. This is equivalent to taking four sum of products. The final result is accumulated in four 16-bit fixed point numbers which are concatenated into a single 64-bit float particle.

`VISFFTCx` A radix-2 FFT of a complex input. The radix-2 decimation-in-time decomposes the overall FFT operation into a series of smaller FFT operations. The smallest operation is the “FFT butterfly” which consists of a single addition and subtraction. Graphically, the full decomposition can be viewed as N stages of FFT butterflies with twiddle factors between each of the stages. One standard implementation is to use three nested for loops to calculate the FFT. The innermost loop calculates all the butterflies and performs twiddle factor multiplcations within a particular stage; the next outer loop calculates the twiddle factors; and the outermost loop steps through all the stages. In order to take advantage of the 16-bit partitioned multiplications

	and additions, the basic operation of the VIS FFT is actually doing four “FFT butterflies” at once. The implementation is similar to the standard three nested for loops, but the last two stages are separated out. In order to avoid packing and unpacking, the basic operation of the last two stages switches from four to two to eventually just one “FFT butterfly”. After the FFT is taken, the order of the sequence is bit-reversed.
`VISParametricEq`	The user supplies the parameters such as *Bandwidth*, *`Center Frequency`*, and *Gain*. The digital biquad coefficients are quickly calculated based on the procedure defined by Shpak.
`VISPackSh`	Takes four float particles, casts them into four signed 16-bit-fixed point numbers, and packs them into a single 64-bit float-particle. The input float particles are first down cast into 16-bit fixed point numbers. The location of the binary point of the fixed point number can be placed anywhere by adjusting the scale parameter. The fixed point numbers are then concatenated to produce a 64-bit result. The order of the fixed point numbers can be reversed so that the most current input either leads or trails the pack, ie reverse equals `FALSE` produces (x[n],x[n-1],x[n-2],x[n-3]) and reverse equals `TRUE` produces (x[n-3],x[n-2],x[n-1],x[n]).
`VISUnpackSh`	Takes a single 64-bit float particle, unpacks them into four 16-bit fixed point numbers, and casts them into four float particles. The input float particle is first separated into four 16-bit fixed point numbers. Once again, the order of the fixed point numbers can be reversed. The fixed point numbers are then up cast to float particles. The exponent value of each float particle can be adjusted by the scaledown parameter.
`VISStereoIn`	Reads Compact Disc audio format from a file given by *fileName*. The file can be the audio port `/dev/audio`, if supported by the workstation. The star reads *`blockSize`* 16-bit samples at each invocation. The *blocksize* should be a multiple of 4.
`VISStereoOut`	Writes Compact Disc audio format to a file given by *fileName.* The file can be the audio port `/dev/audio`, if supported by the workstation. The star writes *blockSize* 16-bit samples at each invocation. The *blocksize* should be a multiple of 4.
`VISInterleaveIn`	Reads Compact Disc audio format from a file given by *fileName*. The file can be the audio port `/dev/audio`, if supported by the workstation. The star reads *blockSize* 16-bit samples at each invocation. The *blocksize* should be a multiple of 4.
`VISInterleaveOut`	Reads Compact Disc audio format from a file given by *fileName*. The file can be the audio port `/dev/audio`, if supported by the workstation. The star reads *blockSize* 16-bit samples at

	each invocation. The *blocksize* should be a multiple of 4.
`VISStereoBiquad`	A two-pole, two-zero IIR filter.
`VISTkStereoIn`	Just like StereoIn, except that a Tk slider is put in the master-control panel to control the volume.
`VISTkStereoOut`	Just like StereoOut, except that a Tk slider is put in the master-control panel to control the volume and balance.
`VISTkParametricEq`	Just like VISParametricEq, except that a Tk slider is put in the master control panel to control the gain.

14.3.14 An Overview of CGC Demos

Figure 14-14 shows the top-level palette of CGC demos. The demos are divided into categories: basic, multirate, signal processing, multi-processor, sound, Tcl/Tk, BDF,HOF and SDF-CGC wormhole demos[1].Many of the demos in the CGC domain have equivalent counterparts in the SDF or BDF domains. See “An overview of SDF demonstrations” on page 5-51, or “An overview of BDF demos” on page 8-3 for brief descriptions of these demos. Brief

1. In Ptolemy0.6, the SDF-CGC Wormhole icon is not present in the CGC demo palette. The SDF-CGC Wormhole demos can be found in the Mixed Domain demo palette, located in the top level Ptolemy palette at $PTOLEMY/demo/init.pal.

descriptions of the demos unique to the CGC domain are given in the sections that follow.

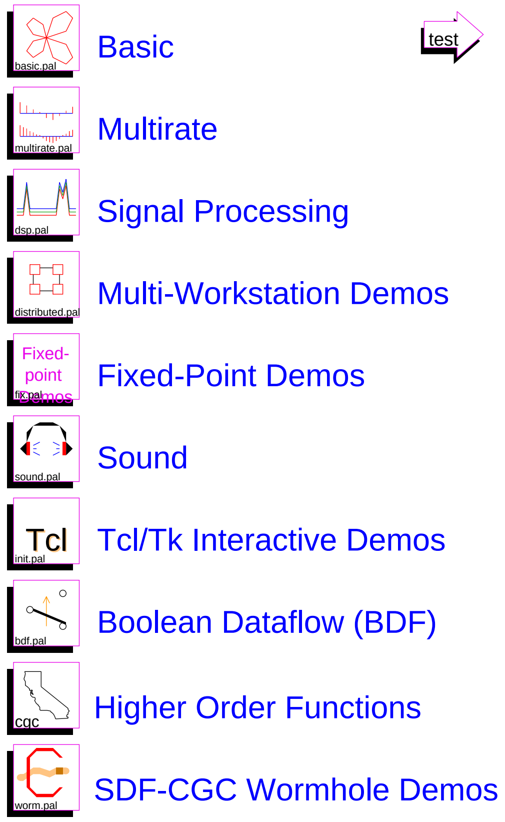


FIGURE 14-14: Top-level palette of demos in the CGC domain

14.3.15 Basic Demos

Figure 14-15 shows the palette of basic demos that are available in the CGC domain. The following demos are equivalent to the SDF demos of the same name (see "An overview of SDF demonstrations" on page 5-51): `butterfly`, `chaos`, `integrator`, `quantize`. The other demos in this palette are described briefly below.

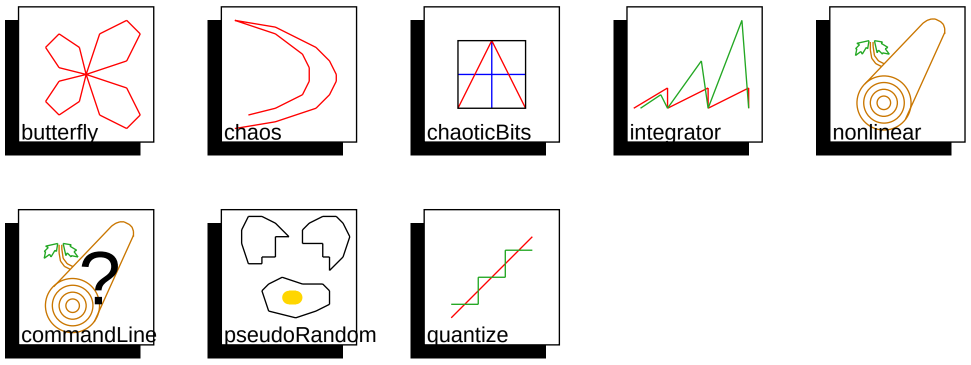


FIGURE 14-15: Basic CGC demos

`chaoticBits`	Chaotic Markov map with quantizer to generate random bit sequence.
`nonlinear`	This simple system plots four nonlinear functions over the range 1.0 to 1.99. The four functions are exponential, natural logarithm, square root, and reciprocal.
`commandline`	This demo is a slight modification of the nonlinear demo. It uses the pragma mechanism to indicate the parameters that are to be made settable from the command-line.
`pseudoRandom`	Generate pseudo-random sequences.

14.3.16 Multirate Demos

Figure 14-16 shows the palette of multirate demos available in the CGC domain. The following demos are equivalent to the SDF demos of the same name (see "An overview of SDF demonstrations" on page 5-51): `interp`, `filterBank`. The other demos in this palette are described briefly below.

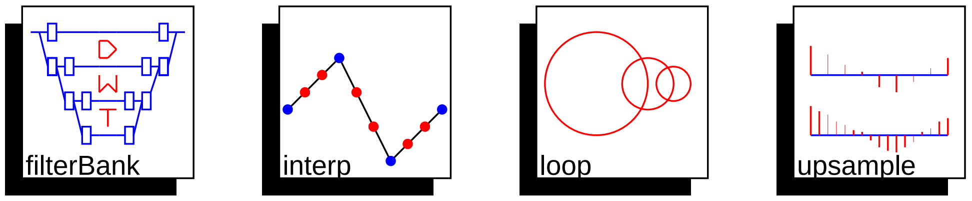


FIGURE 14-16: CGC Multirate demos

`upsample`	This simple up-sample demo tests static buffering. Each invocation of the `XMgraph` star reads its input from a fixed buffer location since the buffer between the `UpSample` star and the `XMgraph` star is static.
`loop`	This demo demonstrates the code size reduction achieved with a loop-generating scheduling algorithm.

14.3.17 Signal Processing Demos

Figure 14-17 shows the palette of signal processing demos that are available in the CGC domain. The following demos are equivalent to the SDF demos of the same name (see "An overview of SDF demonstrations" on page 5-51): `adaptFilter`, `dft`. The `animat-`

edLMS demo is described in “Tcl/Tk Demos” on page 14-24.

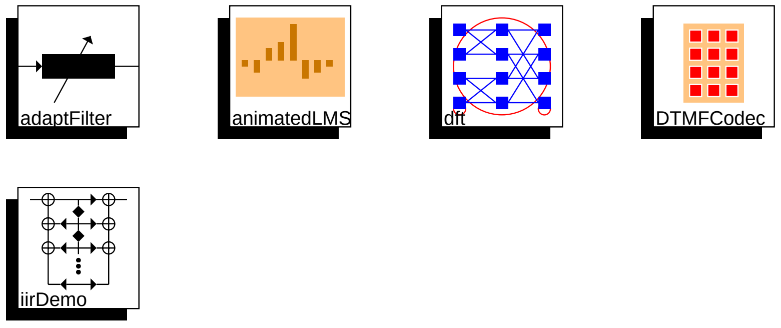


FIGURE 14-17: Signal processing demos in the CGC domain

`DTMFCodec`	Generate and decode touch tones.
`iirDemo`	Two equivalend implementations of IIR filtering. One of the implementations uses the `IIR` star. This demo is not present in Ptolemy0.6.

14.3.18 Multi-Processor Demos

Figure 14-18 shows the top-level palette of multi-processor demos available in the CGC domain. Ptolemy contains two multi-processor targets, `unixMulti_C` and `NOWam`. The demos in each target subpalette are the same. These demos would actually run faster on a single processor, but they do serve as a ‘proof of concept’.

FIGURE 14-18: Multi-processor demos in the CGC domain

Figure 14-19 shows the palette of multi-processor demos that use the `unixMulti_C` target to communicate between workstations.

FIGURE 14-19: Multi-workstation CGC demos

`adaptFilter_multi`	This is a multi-processor version of the `adaptFilter` demo. The graph is manually partitioned onto two networked workstations.
`spread`	This system demonstrates the `Spread` and `Collect` stars. It

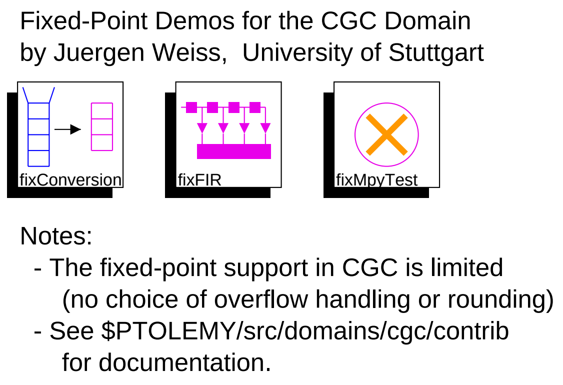
shows how multiple invocations of a star can be scheduled onto more than one processor.

Figure 14-20 shows the demos that use the `NOWam` target to communicate between workstations. The demos in this palette are the same as the demos in the `UnixMulti_C` palette above.

FIGURE 14-20: Networks of Workstations (NOW) CGC demos

14.3.19 Fixed-Point Demos

Figure 14-21 shows the fixed-point demonstrations.



Fixed-Point Demos for the CGC Domain
by Juergen Weiss, University of Stuttgart

Notes:
- The fixed-point support in CGC is limited (no choice of overflow handling or rounding)
- See $PTOLEMY/src/domains/cgc/contrib for documentation.

FIGURE 14-21: Fixed-point demos in the CGC domain

`fixConversion`	Demonstrate fixed-point conversion and overflow effects.
`fixFIR`	Demonstrate tap quantization effects on the transfer function of FIR filters.
`fixMpyTest`	Demonstrate retargeting of a SDF fixed-point multiply demo to CGC.

14.3.20 Sound-Making Demos

Figure 14-22 shows the palette of sound demos available in the CGC domain. Your workstation must be equipped with an audio device that can accept 16-bit linear or μ-law encoded PCM data, for these demos to work.For information about how to use the audio capabilities of a workstation, see “Sounds” on page 2-38.

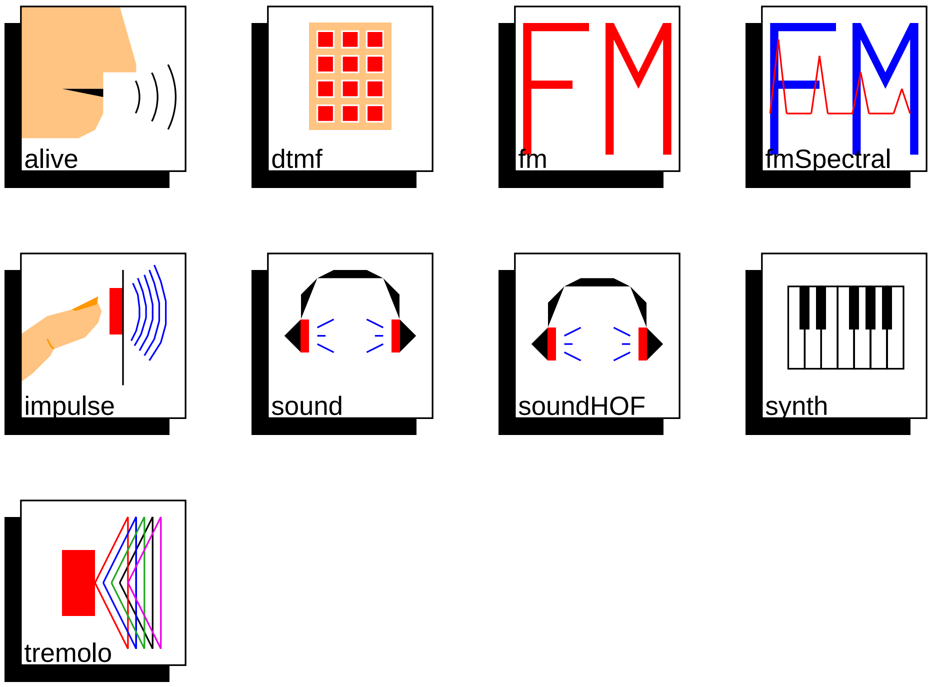


FIGURE 14-22: Sound-making demos in the CGC domain

`alive`	(SGI Only) Processes audio in real time, with an effect similar to the effects Peter Frampton used in the late 70’s rock album ‘Frampton Comes Alive’.
`dtmf`	This demo generates the same dual-tone multi-frequency tones you hear when you dial your telephone. The interface resembles the keypad of a telephone.
`fm`	This demo uses frequency modulation (FM) to synthesize a tone on the workstation speaker. You can adjust the modulation index, pitch, and volume in real time.
`fmSpectral`	FM synthesis with a spectral display.
`impulse`	This demo generates tones on the workstation speaker with decaying amplitude envelopes using frequency modulation synthesis. You can make tones by pushing a button. You can adjust the pitch, modulation index, and volume in real time.
`sound`	Generate a sound to play over the workstation speaker (or headphones).
`soundHOF`	Produce a sound made by adding a fundamental and its harmonics in amounts controlled by sliders.
`synth`	This demo generates sinusoidal tones on the workstation

	speaker. You can control the pitch with a piano-like interface.
`tremolo`	This demo produces a tremolo (amplitude modulation) effect on the workstation speaker. You can adjust the pitch, modulation frequency, and volume in real time.

14.3.21 Tcl/Tk Demos

These demos show off the capabilities of the Tcl/Tk stars, which must be used with the `TclTk_Target` target. Graphical user interface widgets are used to control input parameters and to produce animation. Many of these demos also produce sound on the workstation speaker with the `TkMonoOut` star (see “Tcl/Tk Stars” on page 14-13). Due to the overhead of processing Tk events, you must have a fast workstation (SPARCstation 10 or better) in order to have continuous sound output. You may be able to get continuous sound output on slower workstations if you avoid moving your mouse. Figure 14-23 shows the demos that are available. The following audio demos are documented in the previous section: `dtmf`, `fm`, `audioio,impulse`, `synth`, `tremolo`.

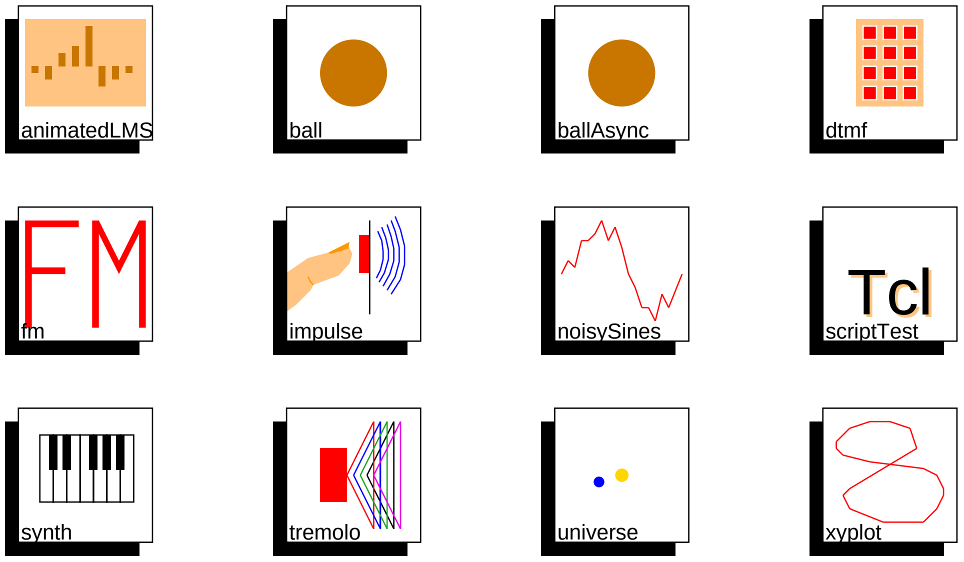


FIGURE 14-23: Tcl/Tk demos in the CGC domain

`animatedLMS`	This demo is a simplified version of the SDF demo of the same name.
`ball`	This demo exhibits sinusoidal motion with a ball moving back and forth.
`ballAsync`	This demo is the same as the `ball` demo except that animation is updated asynchronously.
`noisySines`	Generate a number of sinusoids with controllable additive noise.
`scriptTest`	This demo shows the use of several kinds of Tk widgets for user

	input. Push buttons generate tones or noise, and sliders adjust the frequency and volume in real time.
`universe`	This demo shows the movements of the Sun, Venus, Earth, and Mars in a Ptolemaic (Earth-centered) universe.
`xyplot`	Demonstrate the `TkXYPlot` star.

14.3.22 BDF Demos

Figure 14-24 shows the palette of systems that demonstrate the use of BDF stars in the CGC domain. The `timing` demo is equivalent to the BDF simulation demo of the same name. The demos `bdf-if` and `bdf-doWhile` are equivalent to the BDF simulation demos named `ifThenElse` and `loop`. See “An overview of BDF demos” on page 8-3 for short descriptions of these demos.

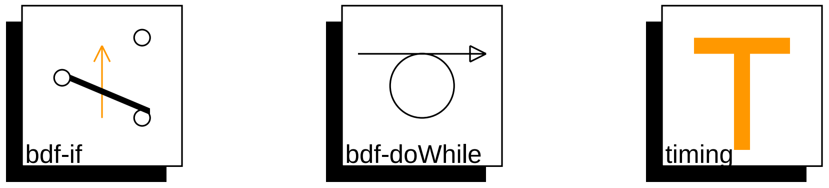


FIGURE 14-24: BDF demos in the CGC domain

14.3.23 Higher Order Function Demos

For information on the HOF demos, see “An overview of HOF demos” on page 6-18.

14.3.24 SDF-CGC Wormhole demos

Figure 14-25 shows that palette of systems that demonstrate the use of the `CreateS-`

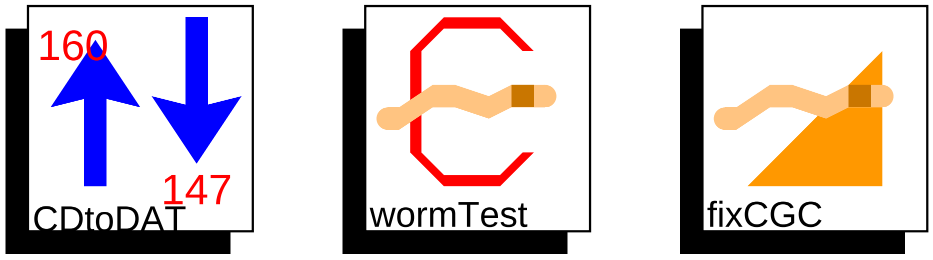


FIGURE 14-25: SDF-CGC Wormhole demos.

`DFStar` CGC target, which allows cgc stars that are reloaded back into Ptolemy for use inside the SDF domain. See “Interface Synthesis between Code Generation and Simulation Domains” on page 13-10 for more information about `CreateSDFStar`. The SDF-CGC Wormhole demos are found under the “Mixed Domain Demos” palette. The Mixed domain Demos palette is in the top level palette that is first visible when pigi starts up.

`CDtoDAT`	Convert two sine waves sampled at CD sample rate to DAT sample rate. The outer galaxy is in the SDF domain, while the

	`cd2dat` galaxy is in the CGC domain. `cd2dat` uses the `CreateSDFStar` target.
`wormTest`	A simple test of the `CreateSDFStar` target.
`fixCGC`	Another simple test of the `CreateSDFStar` target.

14.3.25 UltraSparc VIS Demos

Figure 14-26 show the palette of systems that demonstrate the use of the Sun Ultra-Sparc Visual Instruction Set demos.

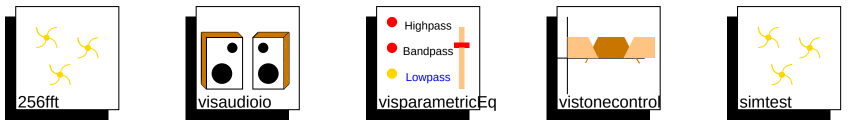


FIGURE 14-26: UltraSparc VIS demos in the CGC domain.

The Visual Instruction Set (VIS) demos only run on Sun UltraSparc workstations with the Sun unbundled CC compiler. The VIS demos will not compile with the Gnu compilers. Note that it is possible to generate VIS code if you don’t have the Sun CC compiler, you just won’t be able to compile it. You must have the Sun Visual Instruction Set Development kit installed, see `http://www.sun.com/sparc/vis/vsdkfaq.html`.

The VIS development kit and the CGC VIS stars require that following two environment variables be set:

```
setenv VSDKHOME /opt/SUNWvsdk
setenv INCASHOME /opt/SUNWincas
```

`256fft`	Plots the real and imaginary parts of a FFT. Note that this demo uses the CGC `Makefile_C` target and sets the *skeletonMakefile* target parameter to a special CGC VIS makefile at `$PTOLEMY/lib/cgc/makefile_VIS.mk`
`visaudioio`	Reads in audio from line-in and plays back from line-out. This demo uses the `makefile_TclTk_VIS.mk` file.
`parametricEQ`	Parametric equalizer.
`vistonecontrol`	Tone control using high, low and bandpass filters.
`simtest`	VIS simulator test universe. This demo illustrates a use of the `CGCVISSim` target. Note that this target is very experimental.

14.3.26 EECS20 demos

The Mixed domain demos palette also contains a palette of demos that were designed for EECS20, “Introduction to Real-Time Systems”. These demos are used in a new lower-division course at UC Berkeley. For more information about this course, see `http://www-inst.eecs.berkeley.edu/~ee20`. The demos in this palette are in the CGC and CG56 domains. Most of these demos run on any Sparcstation with audio output. A few of the demos require an S56X DSP card. At this time, these demos are not documented in this manual, see the individual demos on-line for documentation.

14.3.27 Tycho Demos

These demos demonstrate the use of the TychoTarget to create customized Control Panels. Graphical user interface widgets are used to control input and output parameters and to produce animation. The demos make use of the `TkStereoIn` and `TkStereoOut` (see "Tcl/Tk Stars" on page 14-13) to record and play sound on the workstation speaker, so these demos will probably only work on a Sun Ultrasparc. For information about how to use the audio capabilities of a workstation, see "Sounds" on page 2-38.

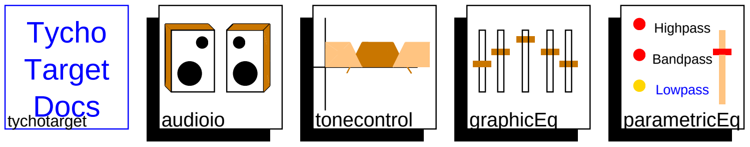


FIGURE 14-27: Tycho Target demos in the CGC domain

`audioio`	This is a simple real time audio demonstration which illustrates Ptolemy's ability to support CD quality audio.
`graphicEq`	This demo consists of 10 band-pass filters with center frequencies spaced out by octaves. Using the customized control panel, you can adjust the gain of each band-pass filter, the record and play volumes and balance in real time.
`parametricEq`	In this demo, there is a single band of parametric equalisation, with control over the band frequency, band width, and band gain. The frequency range is settable; in the future, it will also be possible to select low-pass, band-pass, or high-pass filtering as well.
`tonecontrol`	The demo consists of one of each of the high, band and low-pass filters. There is a single control panel, with control over the band gain for each filter.

Chapter 15. CG56 Domain

Authors: *Joseph T. Buck*
José Luis Pino

Other Contributors: *Brian L. Evans*
Chih-Tsung Huang
Christopher Hylands
Kennard White

15.1 Introduction

The CG56 domain generates assembly code for the Motorola 56000 series of digital signal processors. The graphs that we can describe in this domain follow the synchronous dataflow (SDF) model of computation. SDF allows us to schedule the `Blocks` and allocate all the resources at compile time. Refer to chapter “SDF Domain” on page 5-1 for a detailed description on the properties of SDF.

The Motorola 56000 series are fixed-point digital signal processors. The 56000 and 56001 processors have 24-bit data and instructions, and operate at a maximum clock rate of 40 MIPS. The 56100 processor has 16-bit data and instructions, operates at a maximum rate of 30 MIPS, and has analog/digital and digital/analog converters integrated on the chip. The 56301 has 24-bit data and instructions, operates at a maximum rate of 80 MIPS, and has several built-in input/output interfaces. Although the processors have pipelines of different lengths, the assembly code is backward compatible. The CG56 domain generates assembly code for the 56000 processor and has been tested on the Motorola simulator and on a 56001 board.

Since the 56000 processors are fixed point, the floating point data type has no meaning in the CG56 domain. Fixed-point values can take on the range [-1,1). The most positive value is $1 - 2^{-23}$ for the 56000 and 56300, and $1 - 2^{-15}$ for the 56100. The domain defines a new constant `ONE` set to this maximum positive value. In this chapter, whenever data types are not mentioned, fixed-point is meant. The complex data type means a pair of fixed-point numbers. The complex data type is only partially supported in that it is not supported for stars that have `anytype` inputs or outputs, except for `fork` stars. Integers are the same length as the fixed-point representation. Matrix data types are not supported yet.

Some of the demos use the Motorola 56000 assembler and simulator. You do not need to have a 56000 chip to run the simulator demos, the assemberl and simulator are available for downloading from Motorola at `http://www.mot.com/SPS/DSP/developers/clas.html`.

15.2 An overview of CG56 stars

The “open-palette” command in pigi (`O`) will open a checkbox window that you can use to open the standard palettes in all of the installed domains. For the CG56 domain, the star library is large enough that it has been divided into sub-palettes as was done with the SDF main palette.

The top-level palette is shown in figure 15-1. The palettes are Signal Sources, I/O, Arithmetic, Nonlinear Functions, Logic, Control, Conversion, Signal Processing, and Higher Order Functions. The stars on the Higher Order Functions (HOF) palette are used to help lay out schematics graphically. The HOF stars are in the HOF domain, and not the CG56 domain. The names of the others palettes are modeled after the SDF star palettes of the same name in section 5.2 on page 5-4, except the I/O palette which contains target-specific I/O stars for the Ariel S-56X DSP board and the Motorola 56001 simulator. Each palette is summarized in more detail below. More information about each star can be obtained using the on-line "profile" command (`,`), the on-line man command (`M`), or by looking in the *Star Atlas* volume of *The Almagest*.

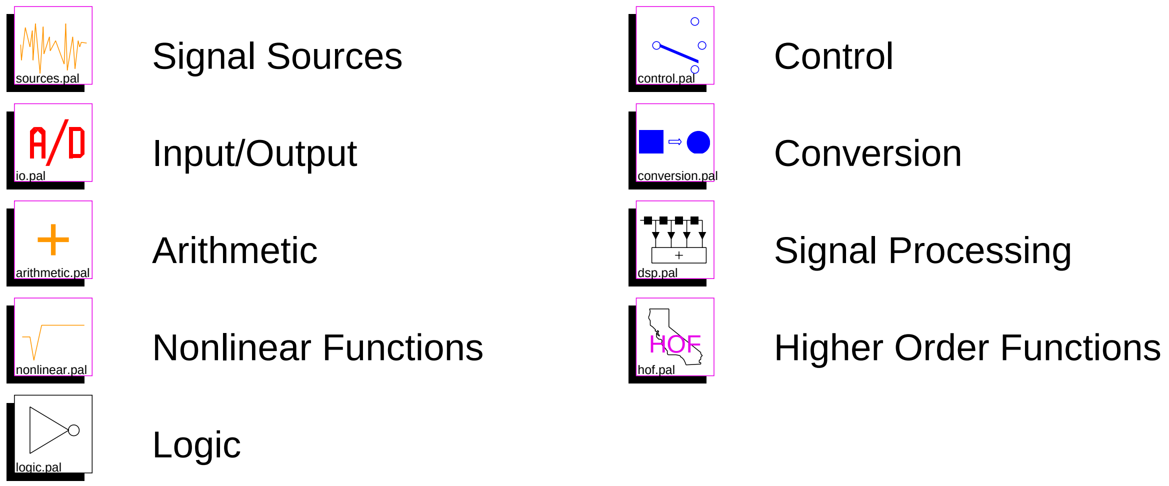


FIGURE 15-1: The palette of star palettes for the CG56 domain.

At the top of each palette, for convenience, are instances of the delay icon, the bus icon, and the following star:

`BlackHole`	Discard all inputs. This star is useful for discarding signals that are not useful.

15.2.1 Source stars

Source stars are stars with only outputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. The palette of source stars is shown in figure 15-2. Refer to 5.2.1 on page 5-5 for descriptions of the SDF equivalent

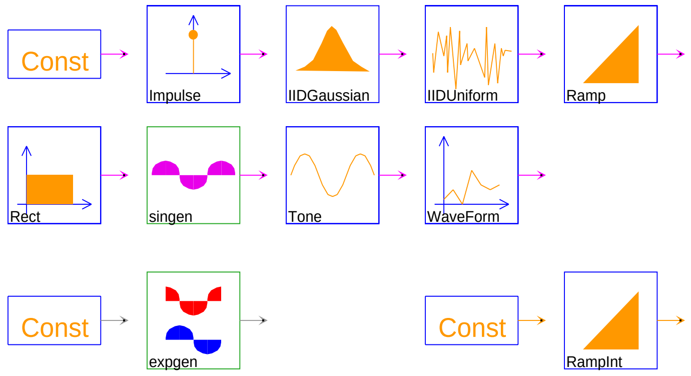


FIGURE 15-2: The palette of source stars for the CG56 domain.

stars: `Const`, `ConstCx`, `ConstInt`, `Ramp`, `RampInt`, `Rect`, `singen`, and `WaveForm`.

`Impulse`	Generate a single impulse of size *impulseSize* (default `ONE`).
`IIDGaussian`	Generate a white Gaussian pseudo-random process with mean 0 and standard deviation 0.1. A Gaussian distribution is realized by summing *noUniforms (default* 16) number of uniform random variables. According to the central limit theorem, the sum of N random variables approaches a Gaussian distribution as N approaches infinity.
`IIDUniform`	Generate an i.i.d. uniformly distributed pseudo-random process. Output is uniformly distributed between -*range* and *range* (default `ONE`).
`Tone`	Generate a sine or cosine wave using a second order oscillator. The wave will be of *amplitude* (default 0.5), *frequency* (default 0.2), and *calcType* (default “sin”)

15.2.2 I/O Stars

I/O stars are target specific stars that allow input and output of stimuli to a target architecture. Currently there are I/O stars for both the Ariel S-56X DSP and the Motorola 56k simulator which are divided hierarchically as shown in figure 15-3.

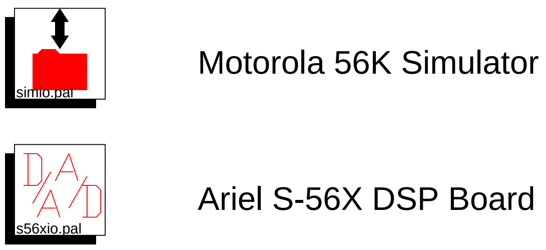


FIGURE 15-3: CG56 I/O Palette

Motorola 56000 Simulator I/O Stars

The palette of I/O stars for the Motorola 56K simulator target is shown in figure 15-4.

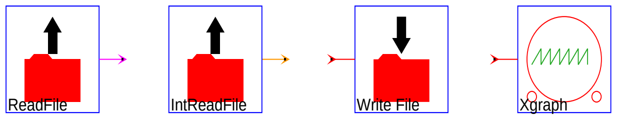


FIGURE 15-4: Motorola Simulator I/O Palette

`ReadFile`	Read fixed-point ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.
`IntReadFile`	Read integer ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.
`WriteFile`	Write data to a file. The simulator dumps the data presented at the input of this star into a specified file.
`Xgraph`	This star shares the same parameters as its SDF and CGC star equivalents. However, with this star, you can only have one input signal. See “pxgraph — The Plotting Program” on page 20-1 to learn about plotting options.

Ariel S-56X DSP Board I/O Stars

The s56xio palette (figure 15-5) allows I/O to the Ariel S-56X DSP board. To use these blocks, you will need access to a S-56X DSP board. These blocks are divided into three sub-categories: generic S-56X, QDM S-56X and CGC-S56X. The QDM stars requires installing qdm, a debugger for DSP systems which was developed by Phil Lapsley at U.C. Berkeley. Qdm is currently available from Mike Peck[1], the designer of the S-56X board.

Generic S-56X

`adjustableGainGX`	Create an interactive adjustable gain using `HostSliderGX`.
`da`	Send the input to both input ports of the SSI star.
`HostAOut`	Output data from the DSP to host via host port asynchronously.
`HostSldrGX`	Generate an athena widget slider for interactive asynchronous input over the host port.
`MagnavoxIn`	Read data from a Magnavox CD player.
`Magnavox`	Read data from and write data to a Magnavox CD player.
`MagnavoxOut`	Write data to a Magnavox CD player.
`PrPrtAD`	Read from the A/D in Ariel ProPort.
`PrPrtADDA`	Read from the A/D and write to the D/A on the Ariel ProPort.To use both the A/D and D/A on a ProPort you must use this star

1. Mike Peck, Berkeley Camera Engineering, mpeck@bcam.com (http://www.bcam.com)

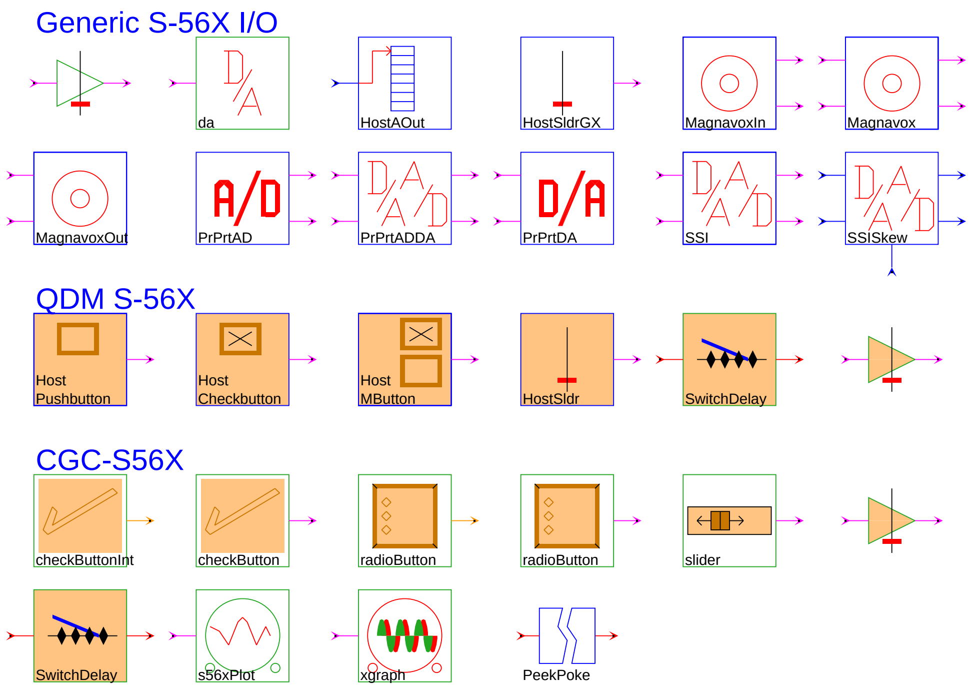


FIGURE 15-5: S-56X I/O Palette

	and not the separate A/D and D/A stars.
`PrPrtDA`	Write to the D/A on the Ariel ProPort.
`SSI`	A generic input/output star for the DSP56001 SSI port.
`SSISkew`	Interface to the 56001 SSI's port with timing-skew capability.

QDM S-56X

To use these stars you must have qdm installed and be using the uniprocessor s-56x target. The target parameter *monitor* must be set to `qdmterm_s56x -run`.

`HostButton`	(2 icons) Graphical two-valued input source. There are two types of buttons: push-buttons and check-buttons. Both present a single button to the user that may be "pressed" with the mouse. The buttons differ in the semantics of the push. When the pushbutton is pressed, the *onVal* state is output, otherwise *offVal*.
`HostMButton`	Graphical one-of-many input source. The star always outputs one of a finite number of values: the output is controlled by the user selecting one of several buttons. Exactly one button in the group is on.
`HostSldr`	Graphical host slider for asynchronous input source.
`SwitchDelay`	This galaxy synchronously switches between the input value and the value of the input delayed by TotalDelay (default 8000) samples.

`adjustableGain`	A user adjustable gain, uses `HostSlider`.

CGC-S56X

`checkButtonInt`	This galaxy creates a Tk checkbutton widget that produces the given *onValue* (default 1) when pressed and *offValue* (default 0) otherwise.
`checkButton`	This galaxy creates a Tk checkbutton widget that produces the given *onValue* (default 1.0) when pressed and *offValue* (default 0.0) otherwise.
`radioButtonInt`	This galaxy creates a Tk radiobutton widget that allows the user to select from among a set of possible output values given by pairs (default “One 1” “Two 2”).
`radioButton`	This galaxy creates a Tk radiobutton widget that allows the user to select from among a set of possible output values given by pairs (default “One 1” “Two 2”)
`slider`	This galaxy creates a Tk slider widget that produces the given value indicated by the slider position which is between low (default 0.0) and high (default 1.0) and initially set to value (default 0.0).
`adjustableGain`	This galaxy multiplies the input by a gain value taken from a Tk slider position between low (default 0.0) and high (default 1.0), which is initially set to value (default 0.0).
`SwitchDelay`	This galaxy synchronously switches between the input value and the value of the input delayed by *TotalDelay* (default 8000) samples.
`s56XPlot`	This galaxy plots the input interactively using TkPlot.
`Xgraph`	This galaxy simply contains a `CGCXgraph` star for use in a CG56 galaxy. The galaxy parameters are identical to those of the enclosed star.
`PeekPoke`	Nondeterminate communication link that splices in a peek/poke pair. In this context, it provides a link between the S-56X Motorola 56001 board and the workstation.

15.2.3 Arithmetic stars

The arithmetic stars that are available are shown in figure 15-6.

`Add`	(2 icons) Output the sum of the inputs. If *saturation* is set to yes, the output will saturate.
`Sub`	Outputs the “pos” input minus all of the “neg” inputs.
`Mpy`	(2 icons) Outputs the product of all of the inputs.
`Gain`	The output is set the input multiplied by a *gain* term. The gain

	must be in [-1,1).
`AddCx`	(2 icons) Output the complex sum of the inputs. If *saturation* is set to yes, the output will saturate.
`SubCx`	Outputs the "pos" input minus all of the "neg" inputs.
`MpyCx`	(2 icons) Outputs the product of all of the inputs.
`AddInt`	(2 icons) Output the sum of the inputs. If *saturation* is set to yes, the output will saturate.
`SubInt`	Outputs the "pos" input minus all of the "neg" inputs.
`MpyInt`	(2 icons) Outputs the product of all of the inputs.
`GainInt`	The output is set the input multiplied by an integer *gain* term.
`DivByInt`	This is an amplifier. The integer output is the integer input divided by the integer *divisor* (default 2). Truncated integer division is used.
`MpyRx`	Multiply any number of rectangular complex inputs, producing an output.
`MpyShift`	Multiply and shift.
`Neg`	Output the negation of the input.
`Shifter`	Scale by shifting left *leftShifts* bits. Negative values of *leftShifts* implies right shifting.

15.2.4 Nonlinear stars

The nonlinear palette (figure 15-7) in the CG56 domain includes transcendental functions, quantizers, table lookup stars, and miscellaneous nonlinear functions.

`Abs`	Output the absolute value of the input.
`ACos`	Output the inverse cosine of the input, which is in the range -1.0 to 1.0. The output, in the principle range of 0 to π, is scaled down by π.

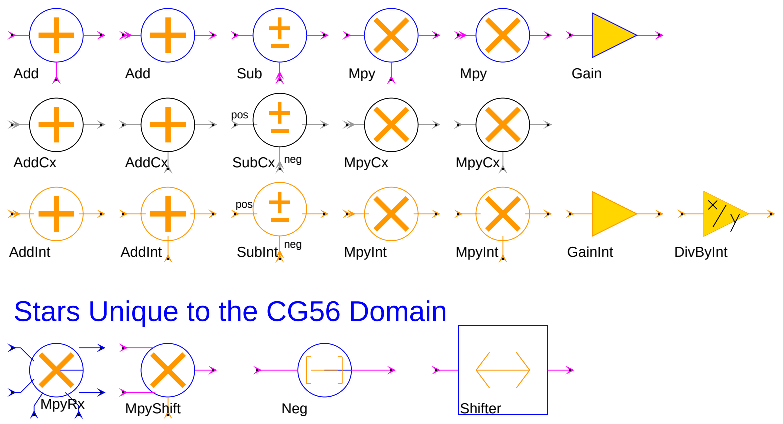


FIGURE 15-6: CG56 Arithmetic Palette

`ASin`	Output the inverse sine of the input, which is in the range -1.0 to 1.0. The output, in the principle range of $-\frac{\pi}{2}$ to $\frac{\pi}{2}$, is scaled down by π.
`Cos`	Output the cosine, calculated the table lookup. The input range is [-1,1] scaled by π.
`expjx`	Output the complex exponential of the input.
`Intgrtr`	An integrator with leakage set by *feedbackGain*. If there is an overflow, the *onOverflow* parameter will designate a wrap around, saturate or reset operation.
`Limit`	Limits the input between the range of [*bottom*, *top*].
`Log`	Outputs the base two logarithm.
`MaxMin`	Output the maximal or minimal (*MAX*) sample out of the last *N* input samples. This can either *compareMagnitude* or take into account the sign. If *outputMagnitude* is `YES` the magnitude of the result is written to the output, otherwise the result itself is written.

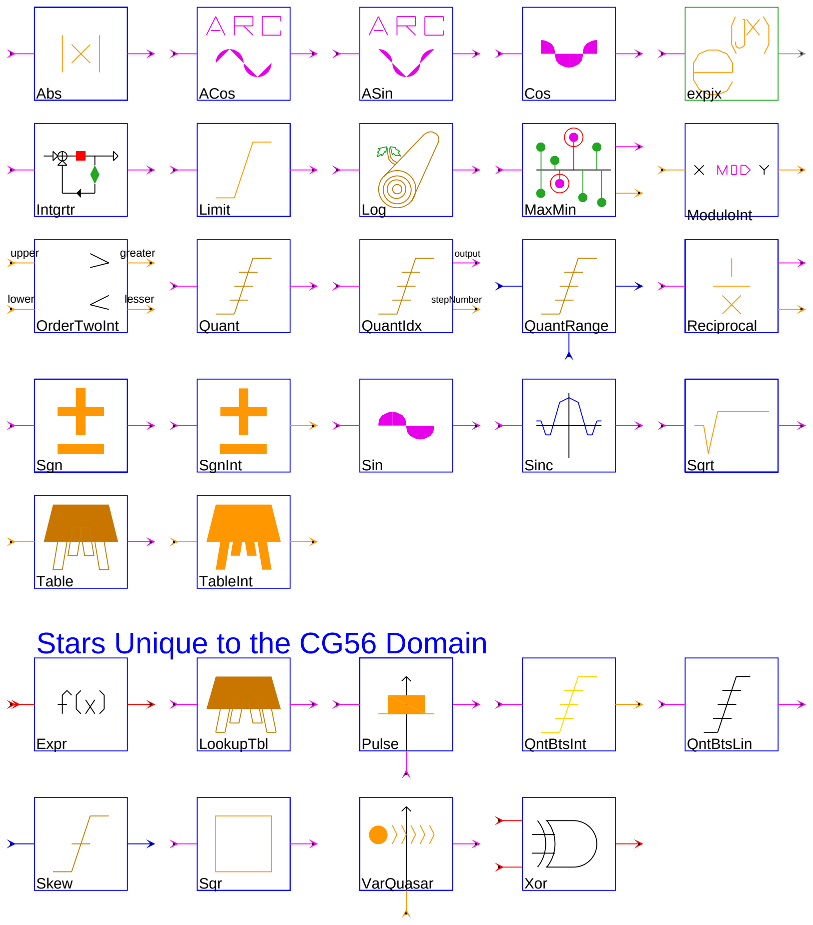


FIGURE 15-7: CG56 Nonlinear Palette

`ModuloInt`	Output the remainder after dividing the integer input by the integer *modulo* parameter.
`OrderTwoInt`	Takes two inputs and outputs the greater and lesser of the two integers.
`Quant`	Quantizes the input to one of N+1 possible output *levels* using N *thresholds*.
`QuantIdx`	The star quantizes the input to one of N+1 possible output *levels* using N *thresholds*. It also outputs the index of the quantization level used.
`QuantRange`	Quantizes the input to one of N+1 possible output *levels* using N *thresholds*.
`Reciprocal`	Outputs the reciprocal to *Nf* precision in terms of a fraction and some left shifts.
`Sgn`	Outputs the sign of the input.
`SgnInt`	Outputs the sign of the integer input.
`Sin`	Outputs the sine, calculated using a table lookup. The input range is [-1,1) scaled by π.
`Sinc`	Outputs the sinc functions calculated as sin(x)/x.
`Sqrt`	Outputs the square root of the input.
`Table`	Implements a real-valued lookup table. The *values* state contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received.
`TableInt`	Implements an integer-valued lookup table. The *values* state contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received.
`Expr`	General expression evaluation.
`LookupTbl`	The input accesses a lookup table. The *interpolation* parameter determines the output for input values between table-entry points. If *interpolation* is "linear" the star will interpolate between table entries; if *interpolation* is set to "none", it will use the next lowest entry.
`Pulse`	Generates a variable length pulse. A pulse begins when a non-zero trigger is received. The pulse duration varies between 1 and *maxDuration* as the control varies between [-1,1).
`QntBtsInt`	Outputs the two's complement number given by the top *noBits* of the input (for integer output).
`QntBtsLin`	Outputs the two's complement number given by the top *noBits* of the input, but an optional *offset* can be added to shift the output levels up or down.

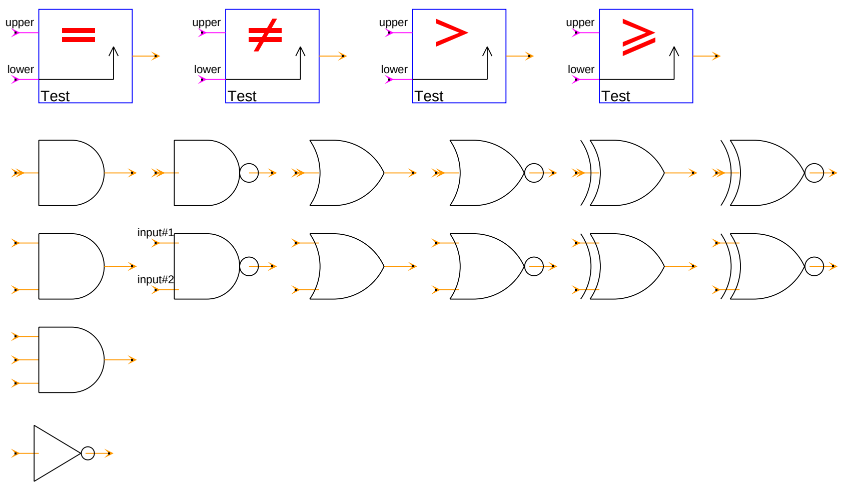


FIGURE 15-8: CG56 Logic Palette

`Skew`	Generic skewing star.
`Sqr`	Outputs the square of the input.
`VarQuasar`	A sequence of values(*data*) is repeated at the output with period N (integer input), zero-padding or truncating the sequence to N if necessary. A value of O for N yields an aperiodic sequence.
`Xor`	Output the bit-wise exclusive-or of the inputs.

15.2.5 Logic stars

The Logic stars are discussed below:

`Test`	(4 icons) Test to see if two inputs are equal, not equal, greater than, and greater than or equal. For less than and less than or equal, switch the order of the inputs.
`And`	(3 icons) True if all inputs are non-zero.
`Nand`	(2 icons) True if all inputs are not non-zero.
`Or`	(2 icons) True if any input is non-zero.
`Nor`	(2 icons) True if any input is zero.
`Xor`	(2 icons) True if an odd number of inputs is non-zero.
`Xnor`	(2 icons) True if an even number of inputs is not non-zero.
`Not`	Logical inverter.

15.2.6 Control stars

Control stars (figure 15-9) manipulate the flow of tokens. All of these stars are polymorphic; they operate on any data type. Refer to 5.2.6 on page 5-17 for descriptions of the SDF equivalent stars: `Fork`, `DownSample`, `Commutator`, `Distributor`, `Mux`, `Repeat`, `Reverse`, and `UpSample`.

`ChopVarOffset`	This star has the same functionality as the `Chop` star except now the *offset* parameter is determined at run time through a control input.
`Cut`	On each execution, this star reads a block of *nread* samples (default 128) and writes *nwrite* of these samples (default 64), skipping the first offset samples (default 0). It is an error if *nwrite* + *offset* > *nread*. If *nwrite* > *nread*, then the output consists of overlapping windows, and hence *offset* must be negative.
`Delay`	A delay star of parameter *totalDelay* unit delays.
`Pad`	On each execution, Pad reads a block of *nread* samples and writes a block of *nwrite* samples. The first *offset* samples have value *fill*, the next *nread* output samples have values taken from the inputs, and the last *nwrite* - *nread* - *offset* samples have value *fill* again.
`Rotate`	The star reads in an input block of a certain *length* and performs

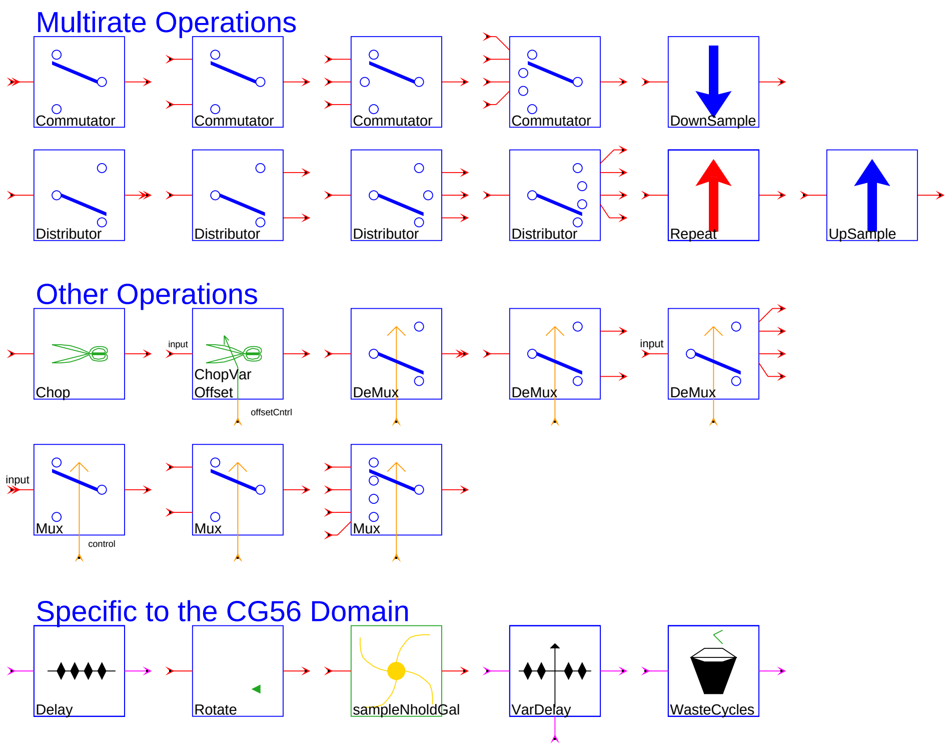


FIGURE 15-9: CG56 Control Palette

	a circular shift of the input. If the *rotation* is positive, the input is shifted to the left so that ouput[0] = input[*rotation*]. If the *rotation* is negative, the input is shifted to the right so that output[*rotation*] = input[0].
`sampleNholdGalaxy`	This sample-and-hold galaxy is more memory efficient than using a downsample star for the same purpose.
`VarDelay`	A variable delay that will vary between 0 and *maxDelay* as the control input varies between -1.0 and 1.0.
`WasteCycles`	Stalls the flow of data for *cyclesToWaste* number of cycles.

15.2.7 Conversion stars

The palette in figure 15-10 shows stars for format conversions from fixed point to complex fixed point. The complex data type is only partially implemented in CG56. Complex

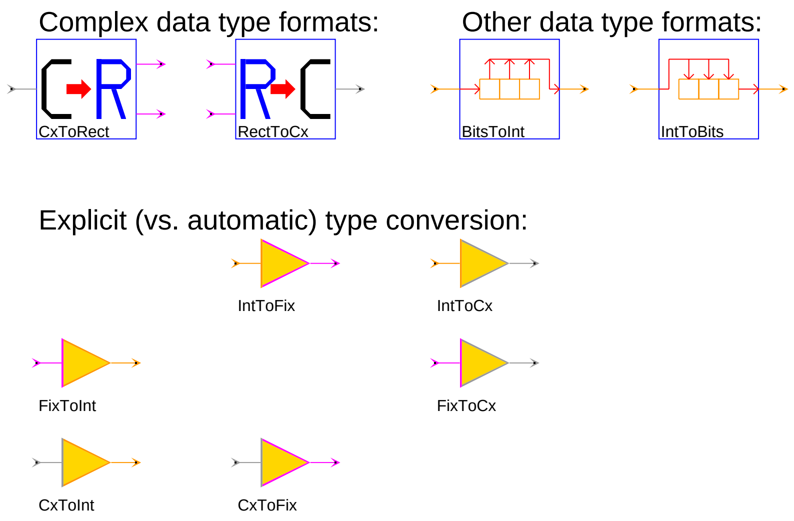


FIGURE 15-10: CG56 Conversion Palette

ports can be connected only to complex ports. Anytype ports can only be connected to fixed and integer ports

`CxToRect`	Output the real part and imaginary part of the input of separate output ports.
`RectToCx`	Output a complex signal with real and imaginary part inputs.
`BitsToInt`	Convert a stream of bits to an integer.
`IntToBits`	Convert an integer into a stream of bits.
`FixToCx`	Convert fixed-point numbers to complex fixed-point numbers.
`FixToInt`	Convert fixed-point numbers to complex fixed-point numbers.
`CxToFix`	Convert fixed-point numbers to complex fixed-point numbers.

`CxToInt`	Convert fixed-point numbers to complex fixed-point numbers.
`IntToFix`	Convert fixed-point numbers to complex fixed-point numbers.
`IntToCx`	Convert fixed-point numbers to complex fixed-point numbers.

15.2.8 Signal processing stars

The palette shown in figure 15-11 has icons for the library of signal processing functions. The filter stars follow. The `Goertzel` and `IIR` stars are identical to their SDF counterparts.

`Allpass` An allpass filter with one pole and one zero. The location of these is given by the “polezero” input.

`Biquad` A two-pole, two-zero IIR filter (a biquad).

$$H(z) = \frac{1 + n_1 z^{-1} + n_2 z^{-2}}{1 + d_1 z^{-1} + d_2 z^{-2}}$$

`Comb` A comb filter with a one-pole lowpass filter in the delay loop.

`BiquadDSPlay` A two-pole, two zero IIR filter (a biquad). This biquad is tailored to use the coefficients from the DSPlay filter design tool. If DSPlay gives the coefficients: A B C D E then define the parameters as follows: a=A, b=B, c=C, d=-(D+1), e = -E. This

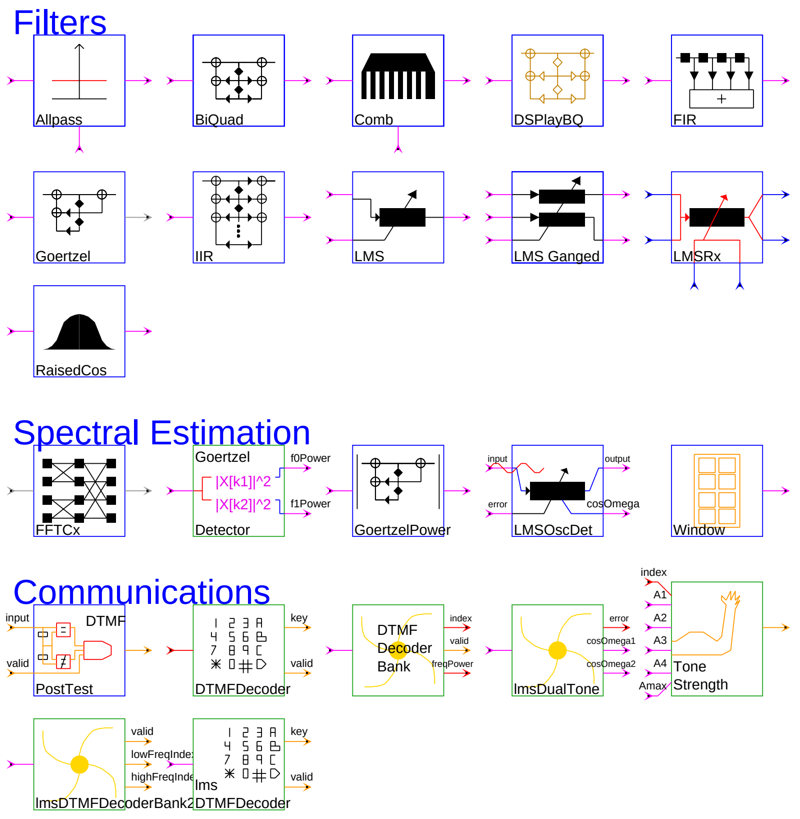


FIGURE 15-11: CG56 Signal processing Palette

only works if a, b, c, d, and e, are in the range [-1,1). The default coefficients implement a low pass filter.

$$H(z) = \frac{a + bz^{-1} + cz^{-2}}{1 - (d+1)z^{-1} - ez^{-2}}$$

`FIR`	A finite impulse response (FIR) filter. Coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear-phase, lowpass filter. To read coefficients from a file, replace the default coefficients with `< filename`, preferably specifying a complete path. Polyphase multirate filtering is also supported.
`LMS`	An adaptive filter using the LMS adaptation algorithm. The initial coefficients are given by the *coef* parameter. The default initial coefficients give an 8th order, linear phase lowpass filter. To read default coefficients from a file, replace the default coefficients with `< filename`, preferably specifying a complete path. This star supports decimation, but not interpolation.
`LMSGanged`	A LMS filter were the coefficients from the adaptive filter are used to run a FIR filter in parallel. The initial coefficients default to a lowpass filter of order 8.
`LMSRx`	A Complex LMS filter
`RaisedCos`	An FIR filter with a magnitude frequency response shaped like the standard raised cosine used in digital communications. See the `SDFRaisedCosine` star for more information.

The spectral estimation stars follow. The `GoertzelDetector`, `GoertzelPower`, and `LMSOscDet` are identical to their SDF counterparts.

`FFTCx`	Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter *order* (default 8) is the transform size. The parameter *direction* (default 1) is 1 for forward, -1 for the inverse FFT.
`Window`	Generate standard window functions or periodic repetitions of standard window functions. The possible functions are `Rectangle`, `Bartlett`, `Hanning`, `Hamming`, `Blackman`, `SteepBlackman`, and `Kaiser`. One period of samples is produced on each firing.

The communications stars are exactly like their SDF counterparts.

15.3 An overview of CG56 Demos

A set of CG56 demonstration programs have been developed. A top-level palette, shown in figure 15-12, contains an icon for each demo palette. The demos are grouped by the CG56 target on which they are implemented. If you do not have the require compiler, simulator, or DSP card, then you can still run the demos to see the generated code. To do this make sure that the *run* and *compile* target parameters are to `NO`. By default, the generated code is written to `$HOME/PTOLEMY_SYSTEMS` directory.

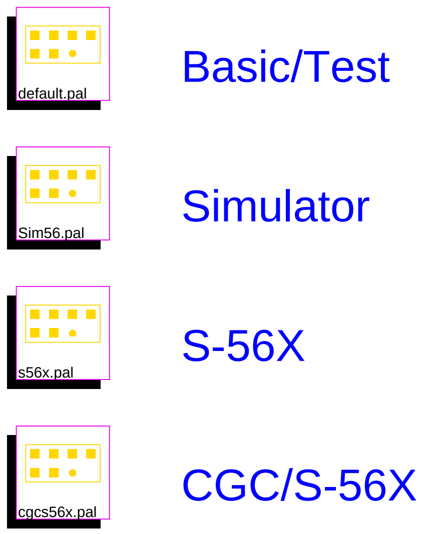


FIGURE 15-12: The top-level demo palette for the CG56

15.3.1 Basic/Test demos

The Basic/Test palette contains six demonstrations (figure 15-13).

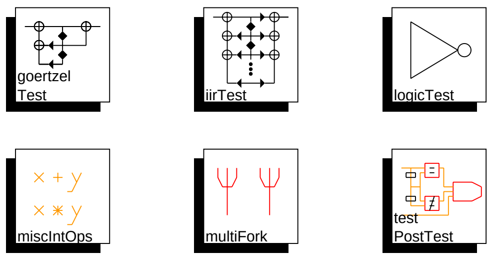


FIGURE 15-13: Basic Demo Palette

`goertzelTest`	Test the Goertzel filters for computing the discrete Fourier transform.
`iirTest`	Test the infinite impulse response (IIR) filters.
`logicTest`	Test various comparison tests and Boolean functions.
`miscIntOps`	Test integer arithmetic operations.
`multiFork`	Test the `AnyAsmFork` star. An `AnyAsmFork` star is one of a group of stars that do produce any code at compile time.
`testPostTest`	Test the `DTMFPostTest` star used in touchtone decoding.

15.3.2 Motorola Simulator Demos

The demos in palette figure 15-14 will generate stand alone applications. These applications will consist of: a shell script to control the simulator and output display programs; a simulator command file; and the assembled code to run on the simulator. The simulator can be run in either an interactive mode or in the background by setting the *interactive* target parameter.

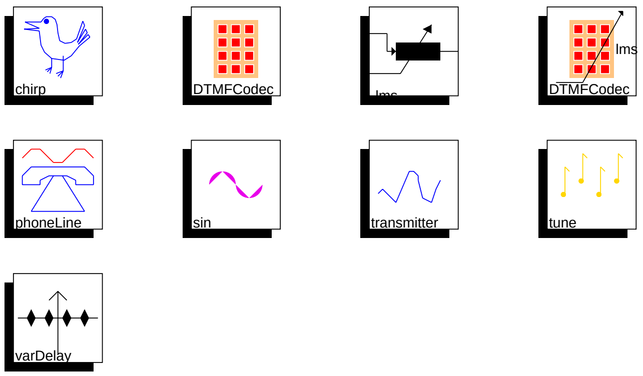


FIGURE 15-14: Motorola Simulator Demos

`chirp`	This system uses two integrators and a cosine to generate a chirp signal.
`DTMFCodec`	Demonstration of touchtone detection using the discrete Fourier transform implemented by using Goertzel filters.
`lms`	A noise source is connected to an eighth-order least-mean squares (LMS) adaptive filter with initial taps specifying a low-pass filter. The taps adapt to a null filter (the impulse response is an impulse) and the error signal is displayed.
`lmsDTMFCodec`	Demonstration of touchtone detection using Normalized Direct Frequency Estimation implemented by using Least-Mean Squares (LMS) adaptive filters.
`phoneLine`	A telephone channel simulator. A tone is passed through some processing which implements various distortions on a telephone channel. The parameters that are controllable are: noise, channel filter, second harmonic, third harmonic, frequency offset, phase jitter frequency, and phase jitter amplitude.
`sin`	A sine wave is generated by using two integrators in a feedback loop.
`transmitter`	A simple 4-level PAM transmitter
`tune`	A tune is generate using FM synthesis of notes stored in a table. The sounds produced are not particularly musically appealing, partly because the modulation index is not variable and the attack and decay profiles are too limited.
`varDelay`	This is a simple application demonstrating variable delay with linear interpolation.

15.3.3 S-56X Demos

The demos shown in figure 15-15 require an Ariel S-56X DSP board to be installed in the workstation. In addition, all but the first demo requires QDM. These demos generate a

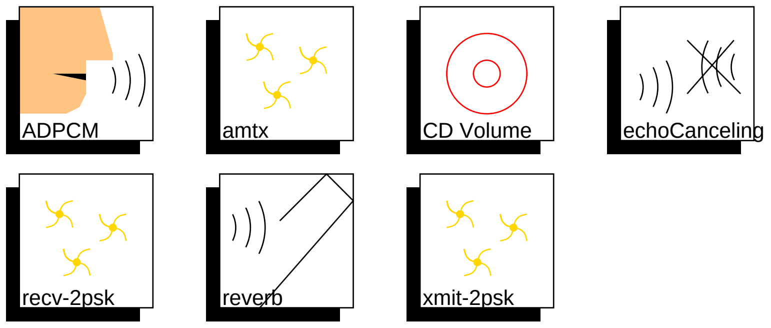


FIGURE 15-15: Ariel S-56X DSP Board demos

stand alone application consisting of: a shell script to download and run the assembled code; a file specifying the asynchronous user I/O interface; and the assembled code.

`ADPCM`	This demo implements a ADPCM coder and decoder. The user at run time can vary the number of quantization bits, the quantization range, and a delay so that signal can be heard instantaneously or a second later. Requires an Ariel Proport and a microphone.
`amtx`	Amplitude Modulation Transmitter. The results of the transmitter are displayed asynchronously at run time.
`CD Volume`	A universe showing a implementing a volume control with `CG56HostSliderGX` stars. Requires a modified CD player.
`echoCanceling`	A system implementing a pair of echo cancellation filters. The first echo cancellation filter cancels an artificial echo introduced by an FIR filter. The second echo cancellation filter is used to cancel the echoes produced by have one microphone next to loud speaker. Another microphone is used for desired input, such as speech. Requires an Ariel Proport and two microphones.
`recv-2psk`	2-PSK Bandpass filter.
`reverb`	This system implements a reverberation system using Comb filters. Requires an Ariel Proport and a microphone.
`xmit-2psk`	2-PSK transmitter.

15.3.4 CGC-S56X Demos

All of the demos in this palette use the `CompileCGSubsystems` target described in section 13.4 on page 13-10.

Stand alone Application Demos

The first set demos generate stand alone applications consisting of two parts: a program generate in C that implements the sub-graph that runs on the host, and a program generated in Motorola 56k assembly that is to be run on the S-56X. The C program initializes and downloads the S-56X program automatically. The first two of the demos shown in figure 15-16, `lms`, `phoneLine`, `DTMFCodec` and `lmsDTMFCodec` are identical to the simulator demos.

`Modem`	The modem palette contain 3 phased shift keying modem demos. These demos illustrate the use of peek/poke actors and hierarchical scheduling. Requires an Ariel Proport and a microphone.
`dtmfSpectrum`	This demos implements a DTMF tone generator and displays the resultant frequency spectrum.
`synth`	A FM music synthesis demonstration. Requires an Ariel Proport.
`synthFFT`	A FM music synthesis demonstration showing the resultant frequency spectrum. Requires an Ariel Proport.
`PRfilterBank`	A perfect reconstruction filter bank.
`ADPCM`	This demo implements a ADPCM coder and decoder. The user at run time can vary the number of quantization bits, the quantization range, and a delay so that signal can be heard instantaneously or a second later. Requires an Ariel Proport and a microphone.

Simulation SDF-Wormhole Demos

The simulation SDF wormhole demos create simulation SDF stars in ptlang and also a load file for the S-56X card. Unlike the other CG56 demos, the applications produced here will not run as stand alone applications. The wormhole allows the user to imbed a CG56 sys-

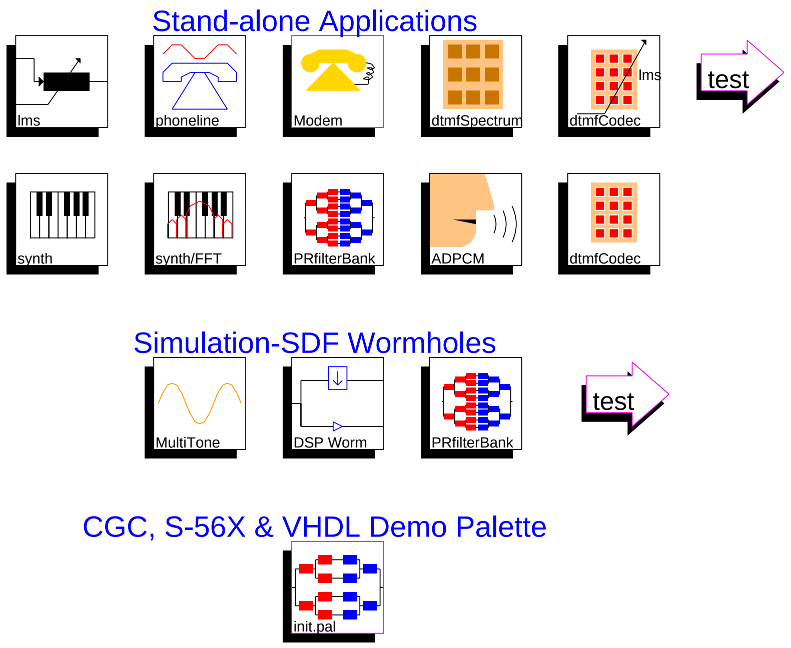


FIGURE 15-16: CGC S-56X demos

tem running on a Ariel S-56X DSP board into a Ptolemy simulation.

`MultiTone`	Generates three sine waves on the S-56X which are at different rates relative to one another.
`DSPWorm`	Demonstrates multirate I/O between Ptolemy and the S-56X board.
`PRfilterBank`	A perfect reconstruction filter bank.

CGC, S-56X & VHDL Demos

The demos in this palette all implement some for of a perfect reconstruction filter bank. One of the examples generates a simulation SDF star which makes use of a VHDL simulator, the S-56X DSP card and the workstation.

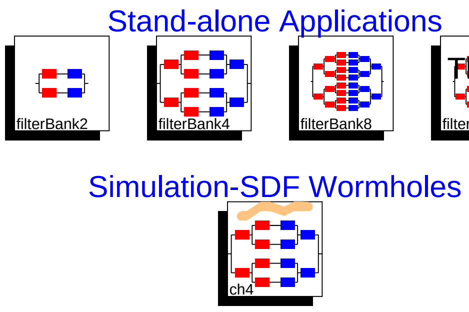


FIGURE 15-17: Combined CGC, CG56, VHDL demos

15.4 Targets

Seven CG56 targets are included in the Ptolemy distribution. To choose one of these targets, with your mouse cursor in a schematic window, execute the Edit:edit-target command (or just type “T”). You will get a list of the available `Targets` in the CG56 domain. The `default-CG56` target is the default value. When you click `OK`, the dialog box appears with the parameters of the target. You can edit these, or accept the defaults. The next time you run the schematic, the selected target will be used.

15.4.1 Default CG56 (default-CG56) target

The default target is used only for code generation. It has the following set of options:

host	(`STRING`) Default = The default is the empty string. Host machine to compile or assemble code on. All code is written to and compiled and run on the computer specified by this parameter. If a remote computer is specified here then `rsh` commands are used to place files on that computer and to invoke the compiler. You should verify that your .rhosts file is properly configured so that `rsh` will work.
directory	(`STRING`) Default = `$HOME/PTOLEMY_SYSTEMS` This is the directory to which all generated files will be written to.
file	(`STRING`) Default = The default is the empty string. This represents the prefix for file names for all generated files.
Looping Level	Specifies if the loop scheduler should be used. Please refer to

the section “default-CG” on page 13-2 for more details on this option. Refer to “Default SDF target” on page 5-65 and “The loop-SDF target” on page 5-67 for more details on loop scheduling.

display?	(`INT`) Default = `YES` If this flag is set to `YES`, then the generated code will be displayed on the screen.
compile?	This is a dummy flag since the default target only generates code.
run?	This is a dummy flag since the default target only generates code.
xMemMap	(`STRING`) Default = `0-4095` Valid x memory address locations. Default is `0-4095`, which means x:0 through x:4095 are valid memory addresses. Disjoint segments of memory can be specified by separating the contiguous ranges with spaces, e.g. “0-4095 5000-5500.”
yMemMap	(`STRING`) Default = `0-4095` Valid y memory address locations. Default is `0-4095`, which means y:0 through y:4095 are valid memory addresses.
subroutines?	(`INT`) Default = `-1` Setting this parameter to N makes the target attempt to generate a subroutine instead of in-line code for a star if the number of repetitions of that star is greater than N (use N=0 to generate subroutines even for stars with just 1 repetition). Set “*subroutines?*” to `-1` (or any other negative integer) to disable the feature.
show memory usage?	(`INT`) Default = `NO` If `YES`, then the target will report the actual amount of program, X data memory, and Y data memory used by the program in words.

15.4.2 CG56 Simulator (sim-CG56) target

This target is used for generating DSP56000 assembly code, assembling it, and running it on a Motorola DSP56000 simulator. For this to work properly, the Motorola 56000 assembler (asm56000) and the simulator (sim56000) must be in the user path. Otherwise a run on this target produces code only, and an error message will appear indicating the absence of the required programs in the user path. Input and output files specified in `ReadFile` and `WriteFile` stars are passed on to the simulator by an automatically generated `universe.cmd` file, which is sourced by the simulator.

The options for this target are mostly the same as the ones for `default-CG56` above, except for the following:

compile?	(`INT`) Default = `YES`

	If this option is set to `YES`, then generated code is assembled using `asm56000` program.
run?	(`INT`) Default = `YES` If `YES`, then the assembled code is run on the Motorola simulator sim56000.
Interactive Sim.	(`INT`) Default = `YES` If `YES` the simulator is run interactively (in which case one can add breakpoints, single step through code, etc.)

15.4.3 Ariel S-56X (S-56X) target

This target generates stand alone applications that will run on the Ariel S-56X DSP board. An optional graphical debugger, QDM, is available from the board designer, Mike Peck. This debugger is needed for some of the user I/O stars that are specific to this target.

The options for this target are mostly the same as the ones for `default-CG56`, except for the following:

monitor	(`STRING`) Default = The default is the empty string.This parameter specifies an optional monitor of debugger for use with the S-56X target. If the application has QDM stars, this parameter should be set to `qdmterm_s56x -run`.

15.4.4 CG56 Subroutine (sub-CG56) target

This target is used to generate subroutines that can be called from hand-written 56000 code. The options are identical to those of `default-CG56` target.

15.4.5 Multiprocessor 56k Simulator (MultiSim-56000) target

This target generates code for a multiprocessor DSP system, where the processors communicate via shared memory. Unfortunately the multiprocessor simulator is not available outside of U.C. Berkeley.

The options for this target are mostly the same as the for `CGMultiTarget`, except for the following:

sMemMap	(`STRING`) Default = `4096-4195` Specifies the shared memory map to use for the communication stars.

Chapter 16. VHDL Domain

Authors: *Michael C. Williamson*

Other Contributors: *Christopher Hylands*
Edward A. Lee
José Luis Pino
William Tsu

16.1 Introduction

The VHDL domain generates code in the VHDL (VHSIC Hardware Description Language) programming language. This domain supports the synchronous dataflow model of computation. This is in contrast to the VHDLB domain, which supports the general discrete-event model of computation of the full VHDL language.

Since the VHDL domain is based on the SDF model, it is independent of any notion of time. The VHDL domain is intended for modeling systems at the functional block level, as in DSP functions for filtering and transforms, or in digital logic functions, independent of implementation issues.

The VHDL domain replaces the VHDLF domain. It is not, however, meant to be used in the same way as the VHDLF domain: the VHDL domain is for generating code from functional block diagrams with SDF semantics, while the VHDLF domain was intended to contrast with the VHDLB domain. It supported structural code generation using VHDL blocks with no execution delay or timing behavior, just functionality. The semantics for the VHDLF domain were not strictly defined, and quite a lot depended on how the underlying VHDL code blocks associated with each VHDLF star were written.

Within the VHDL domain, there are a number of different `Targets` to choose from. The default target, `default-VHDL`, generates sequential VHDL code in a single process within a single entity, following the execution order from the SDF scheduler. This code is suitable for efficient simulation, since it does not generate events on signals. The `SimVSS-VHDL` target is derived from `default-VHDL`, and provides facilities for simulation using the Synopsys VSS VHDL simulator. Communication actors and facilities in the `SimVSS-VHDL` target support code synthesis and co-simulation of heterogeneous CG systems under the `CompileCGSubsystems` target developed by José Pino. There is also a `SimMT-VHDL` target for use with the Model Technology VHDL simulator. The `struct-VHDL` target generates VHDL code in which individual actor firings are encapsulated in separate entities connected by VHDL signals. This target generates code which is intended for circuit synthesis. The `Synth-VHDL` target, derived from `struct-VHDL`, provides facilities for synthesizing circuit representations from the structural code using the Synopsys Design Analyzer toolset. Each of these targets is discussed in more detail in the next section.

Because the VHDL domain uses SDF semantics, it supports retargeting from other domains with SDF semantics (SDF, CGC, etc.) provided that the stars in the original graph are available in the VHDL domain. As this experimental domain evolves, more options for VHDL code generation from dataflow graphs will be provided. These options will include varying degrees of user control and automation depending on the target and the optimization goals of the code generation, particularly in VHDL circuit synthesis.

16.1.1 Setting Environment Variables

In order to have the Synopsys simulation target work correctly, you should make sure that the following environment variables and paths are set correctly. The `SYNOPSYS` and `SIM_ARCH` shell environment variables are settable within the Synopsys simulation target, SimVSS-VHDL, as target parameters by using edit-target (shift-t).

Also, you may need to permanently add the following lines to your .cshrc file and uncomment the ones you wish to take effect:

```
# For VHDL Synopsys demos, uncomment the following:
# setenv SYNOPSYS /usr/tools/synopsys
# setenv SIM_ARCH sparcOS5
# You need the last one of these (.../sge/bin) to run vhdldbx
# since vhdldbx looks for “msgsvr”:
# set path = ( $path $SYNOPSYS/$SIM_ARCH/syn/bin $SYNOPSYS/$SIM_ARCH/
sim/bin $SYNOPSYS/$SIM_ARCH/sge/bin)
# You need this to run vhdlsim, and since vhdldbx calls vhdlsim, you
# need this to run vhdldbx also:
# setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${SYNOPSYS}/${SIM_ARCH}/
sim/lib
#
# For Motorola S56x card demos on the Sparc, you will need something
like:
# setenv S56DSP /users/ptdesign/vendors/s56dsp
# setenv QCKMON qckMon5
# setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${S56DSP}/lib
```

You will need to have a .synopsys_vss.setup file with the right library directive in it in order to use the communication vhdl modules needed for the CompileCGSubsystems target. This file in the root PTOLEMY directory has the correct directive defining the location of the PTVHDLSIM library. Synopsys simulation only sees the file if it is in one of three places: the current directory in which simulation is invoked, the configuration directory within the Synopsys installation tree, or the user’s home directory. Since working directories are frequently created and destroyed, and since the Synopsys installation will vary from site to site, the user’s home directory is the best place to put this file, but each user must do this if the root of their personal Ptolemy tree is anything other than their home directory.

Here is the text in $PTOLEMY/.synopsys_vss.setup:

```
-- This is so communication code can be
-- compiled into the PTVHDLSIM library:
PTVHDLSIM: $PTOLEMY/obj.$PTARCH/utils/ptvhdlsim
```

NOTE: If you build your own tree and it includes your own `$PTOLEMY/src/utils/ptvhdlsim` directory, then you will need to modify your .synopsys_vss.setup file to point to this directory prior to building the new tree. During the build process, this file is needed so that the ptvhdlsim executable can be correctly linked. If it is pointing to some other directory, then you may experience problems linking ptvhdlsim.

16.2 VHDL Targets

The targets of the VHDL domain generate VHDL code from SDF graphs. The targets differ from one another in the styles of VHDL code which they produce, or in the facilities they provide for passing the generated code to VHDL simulation or circuit synthesis tools. The graphs of VHDL actors in Ptolemy are meant to be retargetable in that one graph can be used with multiple VHDL targets, depending on the circumstances. The available targets in the VHDL domain are: `default-VHDL`, `struct-VHDL`, `SimVSS-VHDL`, `SimMT-VHDL`, and `Synth-VHDL`. There is also support for using `SimVSS-VHDL` as a child target of `CompileCGSubsystems` for heterogeneous code generation and co-simulation.

All of the VHDL targets share the following parameters, which are inherited from the base class `HLLTarget`:

directory	(`STRING`) Default = `$HOME/PTOLEMY_SYSTEMS` The name of the directory into which generated code files and supporting files are written. In derived targets, this is also the directory in which compilation for simulation and synthesis are performed.
Looping Level	(`INT`) Default = `0` The control for selecting the looping complexity of the SDF scheduler which is used. Note that looping of code is not supported in the current implementation, except at the main iteration loop on the outside. Therefore a looping level of zero should be used with all loop schedulers or incorrect code may result. In future releases, higher looping levels will be supported.
display?	(`INT`) Default = `TRUE` Option to display generated codefiles to the screen.
write schedule?	(`INT`) Default = `FALSE` Option to write the schedule to a file. The name of the file will be <*galaxy name*>`.`*sched*`.`

16.2.1 The `default-VHDL` Target

The `default-VHDL` target generates VHDL code in a simple and straightforward style which is designed to preserve the SDF scheduling order while incurring minimum VHDL simulation overhead. The code is generated as a single VHDL entity containing a single process of sequential statements. The sequential process reflects the order of execution determined by the SDF scheduler. All data values are stored and communicated through internal variables so that the simulation overhead of VHDL signals and the VHDL discrete-event scheduler can be avoided. No actual simulation is performed by the `default-VHDL` target. It is left to derived targets to support VHDL simulation.

To generate the code, the `default-VHDL` target first invokes the SDF scheduler, and then goes through the resulting schedule in order, firing each VHDL star in sequence. As each VHDL star is fired, a block of VHDL sequential statements is generated. `Porthole` and `State` references and values are resolved and any necessary VHDL variables are created and placed in the list of declared variables. One VHDL star may be fired multiple times and each firing will cause a new codeblock with new variables to be generated. The target manages the communication of data from one VHDL star to the next through VHDL variables. The target also manages state propagation from one firing to the next of the same VHDL star through VHDL variables. State values and tokens remaining on arcs at the end of the schedule iteration are also fed back through the correct variables so that the process can be looped repeatedly and function identically to the original SDF graph.

16.2.2 The `struct-VHDL` Target

The `struct-VHDL` target generates VHDL code in a structural style, in which firings of VHDL stars are individually encapsulated in VHDL entities. The entities are connected to one another through VHDL signals, and the flow of data and state from one firing entity to the next enforces the precedence relationships inherent in the dataflow graph and the resulting schedule. The overall structure of the completed code description parallels the precedence directed acyclic graph (DAG).

The procedure used by the `struct-VHDL` target to generate the code begins similarly to that of the `default-VHDL` target. First, the SDF scheduler is invoked and a valid schedule is computed. Then the schedule is run, and as each VHDL star is fired, the target generates an individual VHDL entity for each firing while keeping track of input and output references to portholes and states. The target manages the references so that it can correctly instantiate each VHDL entity and create VHDL signals to map to the VHDL ports for carrying data and state from one firing to the next. Only firings which have actual dependencies will be connected in the VHDL code representation. In this way, the code generated represents the maximum parallelism in the graph computation outside the granularity level of an individual firing.

The current version of the `struct-VHDL` target also generates registers for latching the values of states and remaining tokens at the end of an iteration. It feeds back the outputs of these registers to the correct inputs at the beginning of the graph so that the structure can be “clocked” by an input clock signal common to all such registers. This clock, on a positive transition, represents the tick of one completed iteration of the dataflow graph. This clock becomes an input to the entire top-level VHDL entity, and will presumably be supplied by an outside source or signal driver during simulation. Similarly, there is an input created for a control signal which selects between the initial values of states or initial tokens and the succeeding values which are passed from one iteration to the next.

16.2.3 The `SimVSS-VHDL` Target

The `SimVSS-VHDL` target is derived from the `default-VHDL` target. It generates code in the same single-entity, single-process, sequential style as the `default-VHDL` target, but it also provides facilities for simulation using the Synopsys VSS VHDL simulator. Depending on the target parameters set when running this target, following the code generation phase this target can compile, elaborate, and execute interactively or non-interactively the design specified by the generated VHDL code.

Communication actors and facilities in the `SimVSS-VHDL` target support code synthesis and co-simulation of heterogeneous CG systems under the `CompileCGSubsystems` target developed by José Pino. This allows a user to manually partition a graph using hierarchy so that multiple codefiles of different code generation domains can be generated. They are then executable if run on host machines which provide all the needed simulators and supporting hardware resources that the individual child targets require. The communication between the different code generation subsystems is automatically generated and correct synchronization and deadlock avoidance are guaranteed. This capability is demonstrated with VHDL in a number of demos included through the main VHDL demo palette.

The additional parameters of the `SimVSS-VHDL` target are as follows:

$SYNOPSYS	(`STRING`) Default = `/usr/tools/synopsys` Value of the `SYNOPSYS` environment variable. It points to the root of the Synopsys tools installation on the host machine.
$ARCH	(`STRING`) Default = `sparcOS5` Value of the `ARCH` environment variable. It indicates which architecture/operating system the Synopsys tools will be run on.
$SIM_ARCH	(`STRING`) Default = `sparcOS5` Value of the `SIM_ARCH` environment variable. It indicates which architecture/operating system the Synopsys VSS simulator will be run on.
analyze	(`INT`) Default = `TRUE` If `TRUE` then attempt to analyze the VHDL code using the `gvan` tool, checking for syntax errors.
startup	(`INT`) Default = `TRUE` If `TRUE` then attempt to startup the VHDL simulator (`vhdldbx` if *interactive* = `TRUE`, else `ptvhdlsim`).
simulate	(`INT`) Default = `TRUE` Currently unused. If *interactive* = `FALSE`, simulation under `ptvhdlsim` will begin automatically following startup.
report	(`INT`) Default = `TRUE` Currently unused.
interactive	(`INT`) Default = `FALSE` If `TRUE` then when simulating, run `vhdldbx`. Otherwise, run `ptvhdlsim`.

16.2.4 The `SimMT-VHDL` Target

The `SimMT-VHDL` target is derived from the `default-VHDL` target. It generates code in the same single-entity, single-process, sequential style as the `default-VHDL` target, and also provides facilities for simulation using the Model Technology VHDL simulator. Depending on the target parameters set when running this target, following the code generation phase this target can compile, elaborate, and execute interactively or non-interactively the design specified by the generated VHDL code.

The additional parameters of the `SimMT-VHDL` target are as follows:

analyze	(`INT`) Default = `TRUE` If `TRUE` then attempt to analyze the VHDL code using the `vcom` tool, checking for syntax errors.
startup	(`INT`) Default = `TRUE` If `TRUE` then attempt to startup the `vsim` VHDL simulator
simulate	(`INT`) Default = `TRUE` Currently unused. If *startup* = `TRUE` and *interactive* = `FALSE`, simulation under `vsim` will begin automatically following startup. If *startup* = `TRUE` and *interactive* = `TRUE`, `vsim` will startup but wait for user input.
report	(`INT`) Default = `TRUE` Currently unused.
interactive	(`INT`) Default = `FALSE` If `TRUE`, then when simulating, start up `vsim` and wait for user input. If `FALSE`, then when simulating, run `vsim` in the background.

16.2.5 The `Synth-VHDL` Target

The `Synth-VHDL` target is derived from the `struct-VHDL` target. It generates code in the same structural style as the `struct-VHDL` target, but it also provides facilities for synthesis and optimization using the Synopsys Design Analyzer toolset.

Not every design which can be specified as an SDF graph using the VHDL stars available in the main star palettes will be synthesizable. Some stars generate code which is not synthesizable under the rules required by the Synopsys Design Analyzer.

There is conceptually more than one way to generate synthesizable VHDL for a given dataflow graph. Just as the sequential VHDL of the `default-VHDL` target differs from the structural VHDL of the `struct-VHDL` target, so there are also multiple ways in which the structural VHDL could be generated. The `struct-VHDL` target as is only generates one particular style. A programmer with some experience could modify this target or create a new or derived target to generate the code in a different structural style to suit different needs. Future releases of Ptolemy may include additional structural VHDL targets for synthesis and/or a modified version of the ones included in the 0.7 release.

The additional parameters of the `Synth-VHDL` target are as follows:

analyze	(`INT`) Default = `TRUE` If `TRUE` then attempt to analyze the VHDL code using the `design_analyzer` tool, checking for syntax errors.
elaborate	(`INT`) Default = `TRUE` If `TRUE` then attempt to elaborate the analyzed design into a netlist form.
compile	(`INT`) Default = `TRUE` If `TRUE` then attempt to compile the elaborated design into an optimized netlist.

report	(`INT`) Default = `TRUE` If `TRUE` then generate reports on the compile-optimized designs for area and timing.

16.2.6 Cadence Leapfrog Ptolemy Interface

Xavier Warzee of Thomson-CSF and Michael C. Williamson created an interface for the Cadence Leapfrog VHDL Simulator.

`$PTOLEMY/src/domains/vhdl/targets` contains the Cadence Leapfrog VHDL Target. This target, `SimLF-VHDL`, allows simulation of generated VHDL code with the Leapfrog simulator from Cadence. This target is analogous to the `SimVSS-VHDL` target, which supports simulation with the Synopsys VHDL System Simulator.

Setup

To use the Leapfrog you need to have the following setup. Locally, our Cadence installation is at `/usr/eesww/cadence`, so your `.cshrc` would contain:

```
setenv PATH /usr/eesww/cadence/9504/tools/leapfrog/bin:$PATH
setenv CDS_LIB /usr/eesww/cadence/9504/tools/leapfrog
setenv CDS_INST_DIR /usr/eesww/cadence/9504
```

You also need to set up some files.

In the directory where the VHDL code is generated, for example `~/PTOLEMY_SYSTEMS/VHDL`, the following two files must be provided:

`cds.lib` contains

```
softinclude $CDS_VHDL/files/cds.lib
define leapfrog ./LEAPFROG
define alt_syn $CDS_INST_DIR/lib/alt_syn
```

`hdl.var` contains:

DEFINE WORK leapfrog include $CDS_VHDL/files/hdl.var

and the directory `~/PTOLEMY_SYSTEMS/VHDL/LEAPFROG` must exist

16.3 An Overview of VHDL Stars

The figure below shows the top-level palette of VHDL stars. The stars are divided into categories: sources, sinks, arithmetic functions, nonlinear functions, control, conversion, signal processing, and higher order functions. The higher order function stars are the same ones that are common to all domains and they are not particular to VHDL. Icons for `delay`, `bus`, and `BlackHole` appear in most palettes for easy access. Most of the stars in the VHDL domain have equivalent counterparts in the SDF domain. See “An overview of SDF stars” on page 5-4 for brief descriptions of these stars.

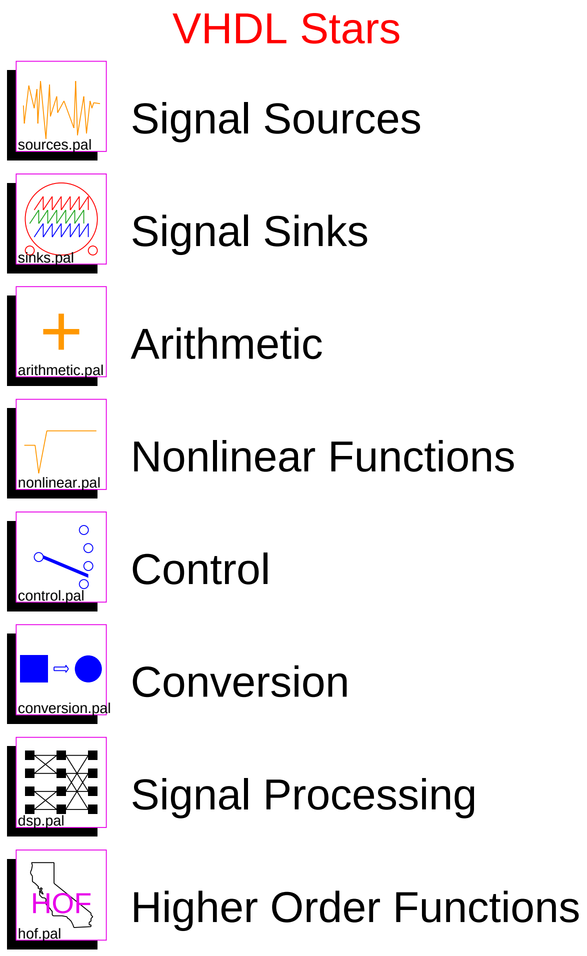


FIGURE 16-1: Top-level palette of stars in the VHDL domain.

16.3.1 Source Stars

Source stars have no inputs and produce data on their outputs. The figure below shows the palette of VHDL source stars. All of these are equivalent to the SDF stars of the same name.

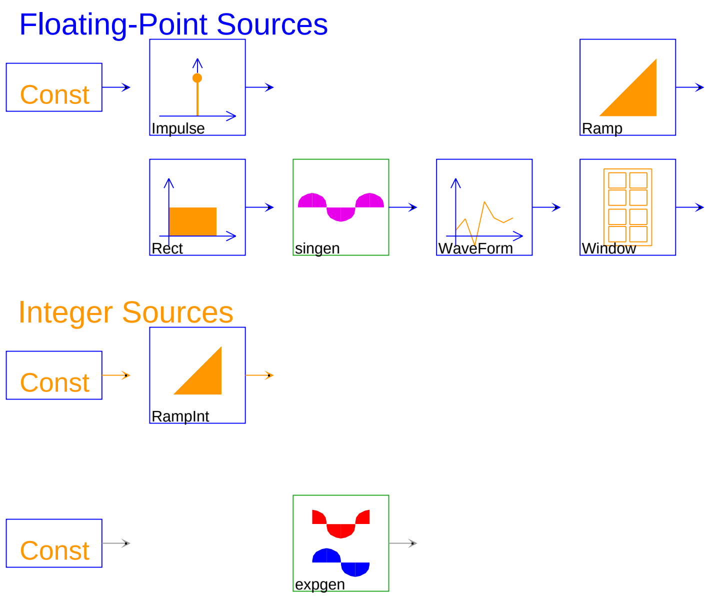


FIGURE 16-2: Source stars in the VHDL domain.

16.3.2 Sink Stars

Sink stars have no outputs and consume data on their inputs. The figure below shows the palette of VHDL sink stars. All of these are equivalent to the SDF stars of the same name.

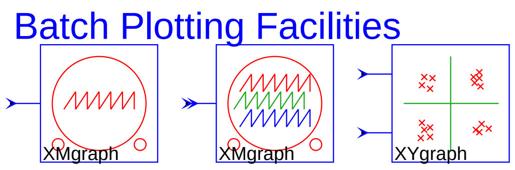


FIGURE 16-3: Sink stars in the VHDL domain.

16.3.3 Arithmetic Stars

Arithmetic stars perform simple functions such as addition and multiplication. The figure below shows the palette of VHDL arithmetic stars. All of the stars are equivalent to the

SDF stars of the same name.

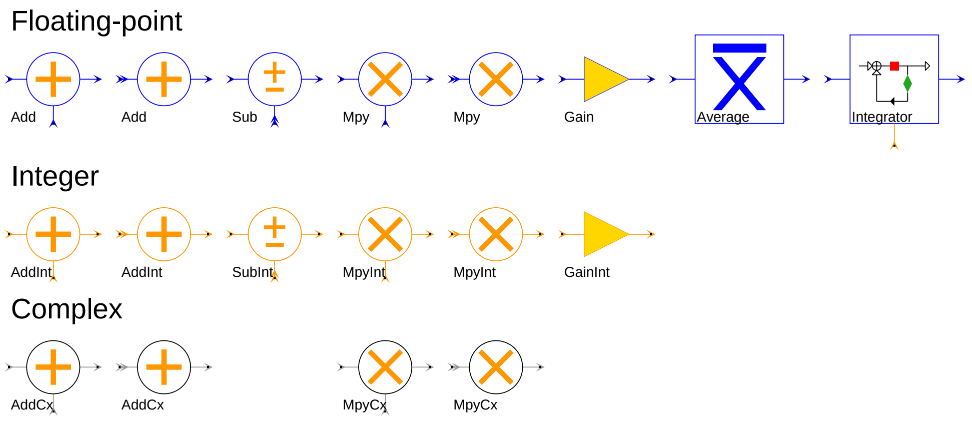


FIGURE 16-4: Arithmetic stars in the VHDL domain.

16.3.4 Nonlinear Stars

Nonlinear stars perform simple functions. The figure below shows the palette of VHDL nonlinear stars. All of these are equivalent to the SDF stars of the same name.

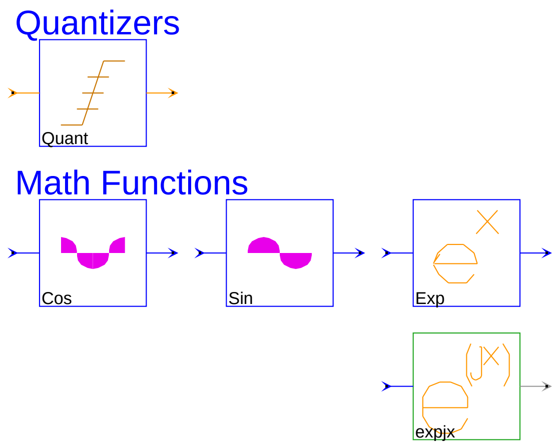


FIGURE 16-5: Nonlinear stars in the VHDL domain.

16.3.5 Control Stars

Control stars are used for routing data and other control functions. The figure below shows the palette of VHDL control stars. All of these are equivalent to the SDF stars of the same name.

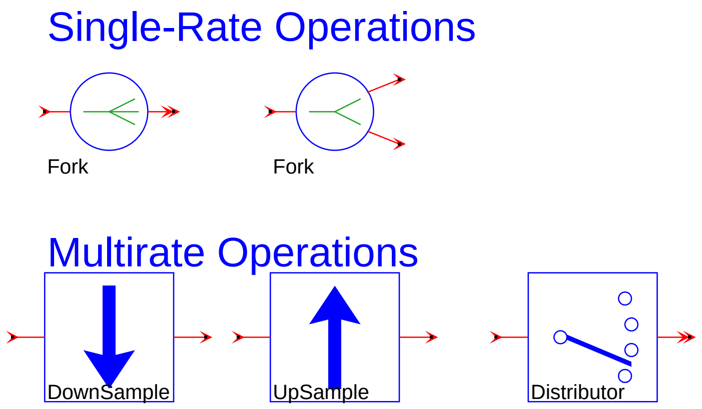


FIGURE 16-6: Control stars in the VHDL domain.

16.3.6 Conversion Stars

Conversion stars are used to convert between different data types. The figure below shows the palette of VHDL conversion stars. All of the stars are equivalent to the SDF stars of the same name.

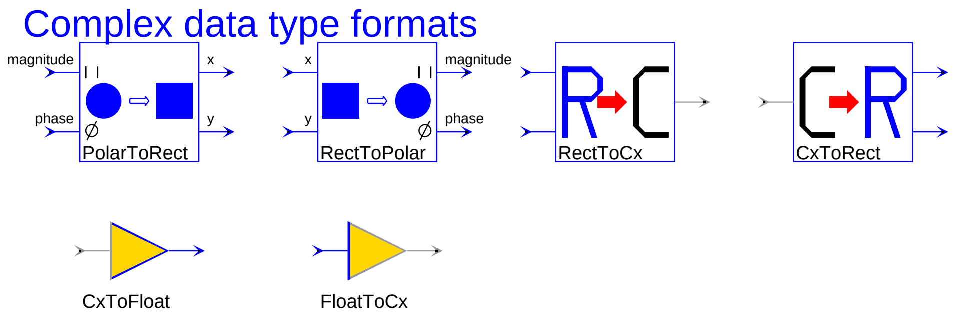


FIGURE 16-7: Type-conversion stars in the VHDL domain.

16.3.7 Signal Processing Stars

The figure below shows the palette of VHDL signal processing stars. All of the stars are equivalent to the SDF stars of the same name (see “Signal processing stars” on page 5-30).

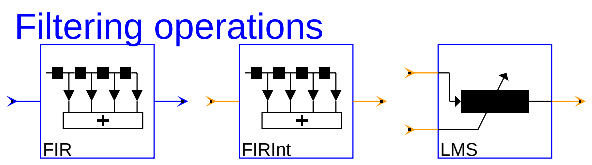


FIGURE 16-8: Signal processing stars in the VHDL domain.

16.4 An Overview of VHDL Demos

The figure below shows the top-level palette of VHDL demos. The demos are divided into categories: code generation, simulation, synthesis, and cosimulation. Some of the demos in the VHDL domain have equivalent counterparts in the SDF or CGC domains. See “An overview of SDF demonstrations” on page 5-51 for brief descriptions of these demos. Brief descriptions of the demos unique to the VHDL domain are given in the sections that follow.

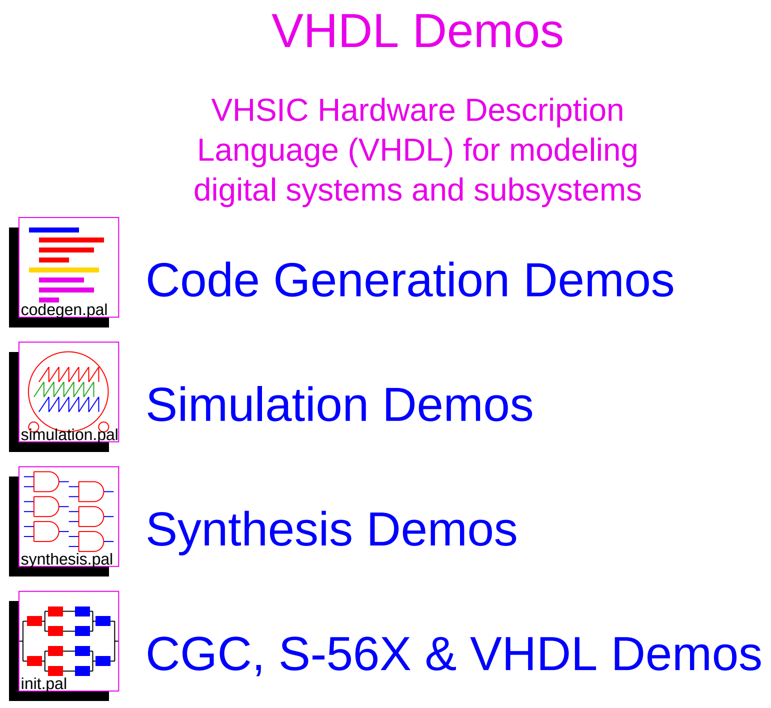


FIGURE 16-9: Top-level palette of demos in the VHDL domain.

16.4.1 Code Generation Demos

Figures below show demos that do nothing but generate code.

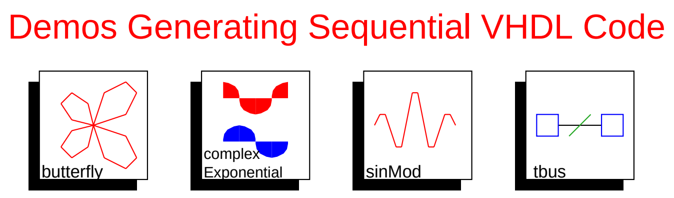


FIGURE 16-10: Sequential Code Generation Demos.

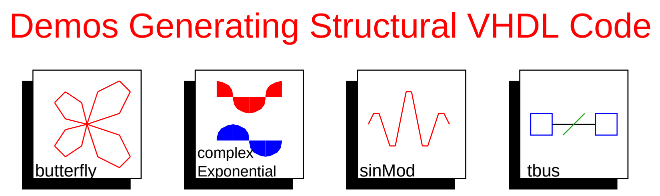


FIGURE 16-11: Structural Code Generation Demos.

The sequential demos use the `default-VHDL` target. The structural demos use the `struct-VHDL` target. They are essentially the same systems being run, but with two different targets producing two different styles of VHDL code. These demos provide a direct comparison of these two basic styles of VHDL code generation.

16.4.2 Simulation Demos

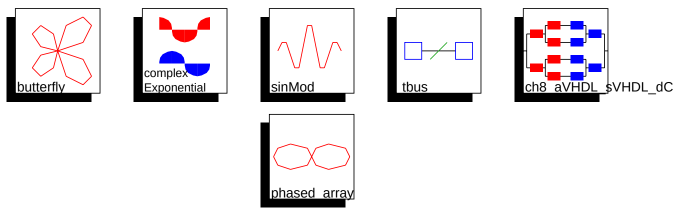


FIGURE 16-12: Demos using the Synopsys VSS Simulator.

These demos use the `SimVSS-VHDL` target. Each one generates VHDL code which is functionally equivalent to the SDF graph specification, and then the code is executed on the Synopsys VSS Simulator. Graphical monitoring blocks provide output analysis of the results of running these systems.

16.4.3 Synthesis Demos

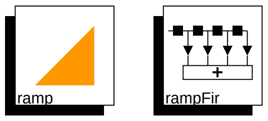


FIGURE 16-13: Demos using the Synopsys Design Analyzer for synthesis.

These demos use the `Synth-VHDL` target. Each one generates structural VHDL code which is equivalent to the SDF specification. One difference is that the data types are converted to simple 4-bit integers to speed up the synthesis process. Once the code is generated, the netlist is synthesized through the Synopsys Design Analyzer. Following that, the netlist is optimized and then control of the Design Analyzer is returned to the user for further exploration and inspection.

16.4.4 Cosimulation Demos

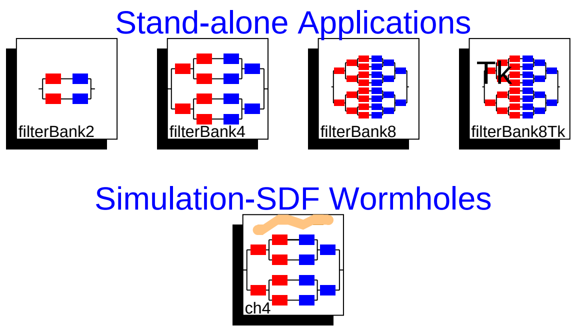


FIGURE 16-14: Demos mixing simulation in VHDL, C, and Motorola DSP56000 code.

These demos use the `CompileCGSubsystems` target which uses the SimVSS-VHDL target as a child target for the VHDL portions of the systems. The first three demos generate stand-alone heterogeneous programs which run in C, Motorola DSP56000 assembly, and VHDL. They produce analysis and synthesis filterbanks for perfect reconstruction using progressively more complex structures. The fourth demo also generates a Tcl/Tk user interface for selecting one of three waveform inputs to the system. The fifth and final demo generates the filterbank system, but instead of doing it as a standalone program, it incorporates the system into a wormhole inside a top-level SDF system. This way the subsystem can be executed in code which is potentially faster than SDF simulation, and it can be reused without having to recompile the subsystem each time the top-level system is executed.

Chapter 17. C50 Domain

Authors: *Luis Gutierrez*

17.1 Introduction

The C50 domain generates assembly code for the Texas Instruments TMS320C5x series of digital signal processors. The graphs that we can describe in this domain follow the synchronous dataflow (SDF) model of computation. SDF allows us to schedule the `Blocks` and allocate all the resources at compile time. Refer to chapter “SDF Domain” on page 5-1 for a detailed description on the properties of SDF.

The TMS320C5x series are fixed-point digital signal processors which have 16 bit data and instructions and operate at a maximum rate of 50MIPS. The C50 domain has been tested on the TMS320C50 DSP Starter Kit board.

Since the C5x processors are fixed point, the floating point data type has no meaning in the C50 domain. Fixed-point values can take on the range [-1,1). The most positive value is $1 - 2^{-15}$. The domain defines a new constant `C50_ONE` set to this maximum positive value. In this chapter, whenever data types are not mentioned, fixed-point is meant. The complex data type means a pair of fixed-point numbers. The complex data type is supported for stars that have `anytype` inputs or outputs. Integers are the same length as the fixed-point representation. Matrix data types are not supported yet.

17.2 An overview of C50 stars

The “open-palette” command in pigi (“O”) will open a checkbox window that you can use to open the standard palettes in all of the installed domains. For the C50 domain, the star library is large enough that it has been divided into sub-palettes as was done with the SDF main palette.

The top-level palette is shown in figure 17-1. The palettes are Signal Sources, I/O, Arithmetic, Nonlinear Functions, Logic, Control, Conversion, Signal Processing, and Higher Order Functions. The stars on the Higher Order Functions (HOF) palette are used to help lay out schematics graphically. The HOF stars are in the HOF domain, and not the C50 domain. Each palette is summarized in more detail below. More information about each star can be obtained using the on-line “profile” command (“,”), the on-line “man” command (“M”), or by looking in the *Star Atlas* volume of *The Almagest*.

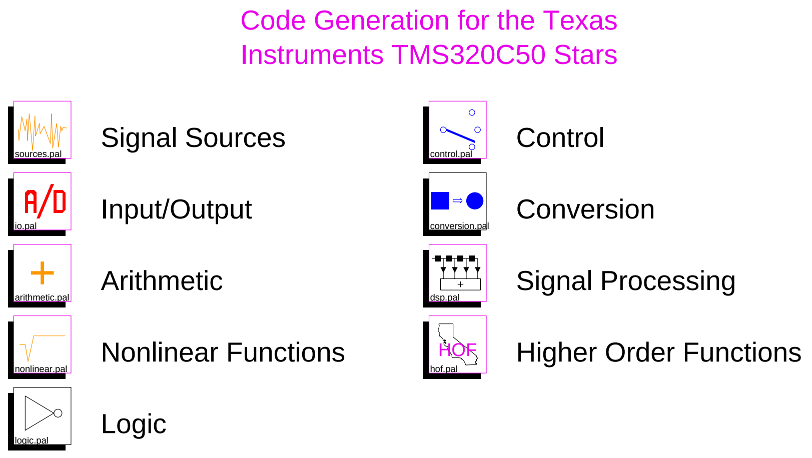


FIGURE 17-1: The palette of star palettes for the C50 domain.

At the top of each palette, for convenience, are instances of the delay icon, the bus icon, and the following star:

`BlackHole`	Discard all inputs. This star is useful for discarding signals that are not useful.

17.2.1 Source stars

Source stars are stars with only outputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. The palette of source stars

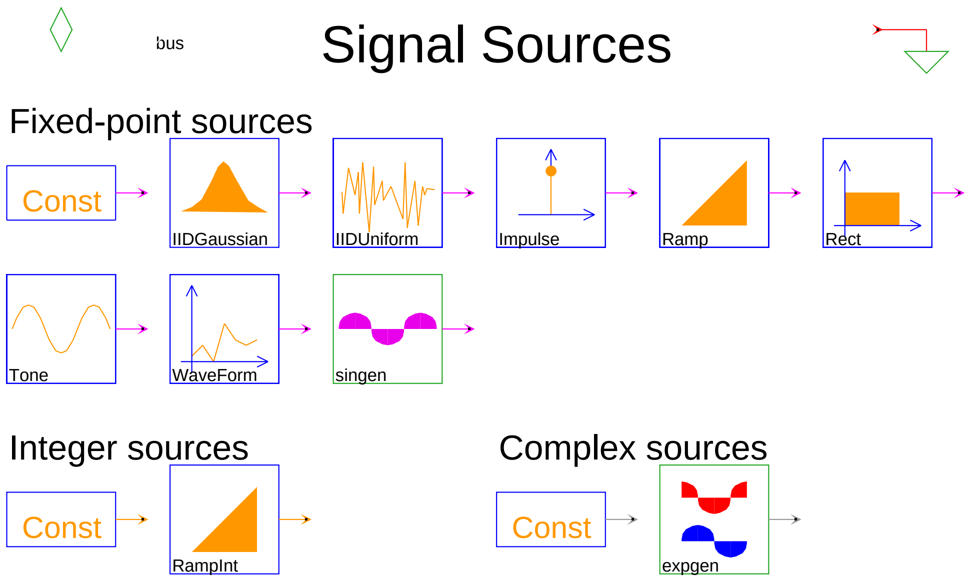


FIGURE 17-2: The palette of source stars for the C50 domain.

is shown in figure 17-2. Refer to 5.2.1 on page 5-5 for descriptions of the SDF equivalent stars: `Const`, `ConstCx`, `ConstInt`, `Ramp`, `RampInt`, `Rect`, `singen`, and `WaveForm`.

`Impulse`	Generate a single impulse or an impulse train. The size is determined by *impulseSize* (default ONE). If *period* (default is 0) is positive an impulse train with this period is generated, otherwise a single impulse is generated. If *delay* (default 0) is positive the impulse (or impulse train) is delayed by this amount.
`IIDUniform`	Generate an i.i.d. uniformly distributed pseudo-random process. Output is uniformly distributed between *-range* and *range* (default ONE).
`IIDGaussian`	Generate a white Gaussian pseudo-random process with mean 0 and standard deviation 0.1. A Gaussian distribution is realized by summing *noUniforms (default* 16) number of uniform random variables. According to the central limit theorem, the sum of N random variables approaches a Gaussian distribution as N approaches infinity.
`Tone`	Generate a sine or cosine wave using a second order oscillator. The wave will be of *amplitude* (default 0.5), *frequency* (default 0.2), and *calcType* (default “sin”)

17.2.2 I/O Stars

I/O stars are target specific stars that allow input and output of stimuli to a target architecture. Currently there are I/O stars only for the C50 DSK board so these stars should only be used with the DSKC50 target. These stars are located on the TI 320C5x IO palette inside the Input/Output palette.

`AIn`	This is an interrupt driven star to receive samples from the A/D converter in the Analog Interface Chip. The sample rate is determined by *sampleRate*. The actual conversion rate is 285.7KHz/N where N is an integer from 4 to 64. This star supports an internal buffer to hold the received samples. The size of this buffer can be set manually by changing the *interruptBufferSize* parameter. Setting *interruptBufferSize* to a negative value will set the size of the buffer equal to the number of times the star is fired on each iteration of the universe.
`AOut`	This is an interrupt driven star to send samples to the D/A converter in the AIC chip. The parameters are identical to those of the `AIn` star.

17.2.3 Arithmetic stars

The arithmetic stars that are available are shown in figure 17-4.

`Add`	(2 icons) Output the sum of the inputs. If *saturation* is set to yes, the output will saturate.
`Sub`	Outputs the “pos” input minus all of the “neg” inputs.
`Mpy`	(2 icons) Outputs the product of all of the inputs.
`Gain`	The output is set the input multiplied by a *gain* term. The gain must be in [-1,1).
`AddCx`	(2 icons) Output the complex sum of the inputs. If *saturation* is set to yes, the output will saturate.
`SubCx`	Outputs the “pos” input minus all of the “neg” inputs.
`MpyCx`	(2 icons) Outputs the product of all of the inputs.
`AddInt`	(2 icons) Output the sum of the inputs. If *saturation* is set to yes, the output will saturate.
`SubInt`	Outputs the “pos” input minus all of the “neg” inputs.

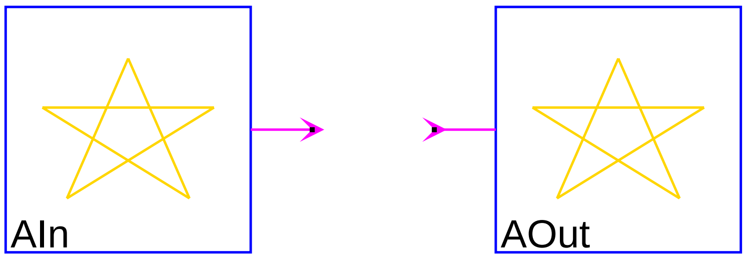


FIGURE 17-3: TI DSK 320C5x IO Palette

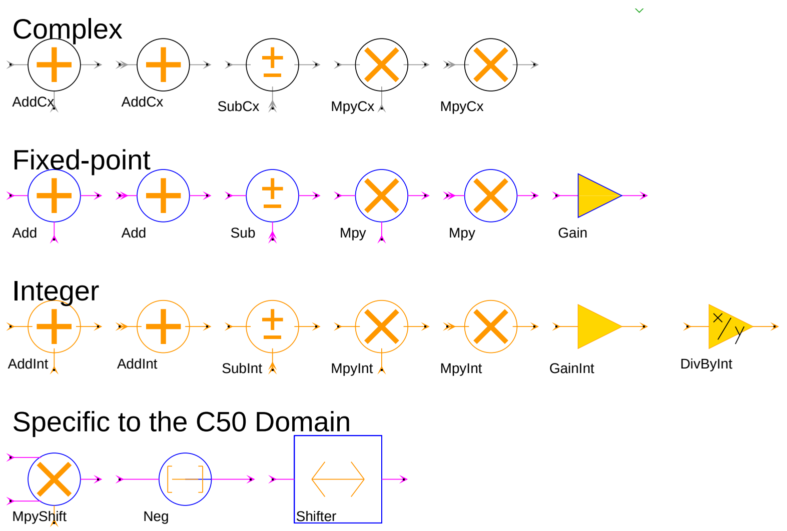


FIGURE 17-4: C50 Arithmetic Palette

`MpyInt`	(2 icons) Outputs the product of all of the inputs.
`GainInt`	The output is set the input multiplied by an integer *gain* term.
`DivByInt`	This is an amplifier. The integer output is the integer input divided by the integer *divisor* (default 2). Truncated integer division is used.
`MpyShift`	Multiply and shift.
`Neg`	Output the negation of the input.
`Shifter`	Scale by shifting left *leftShifts* bits. Negative values of *leftShifts* implies right shifting.

17.2.4 Nonlinear stars

The nonlinear palette (figure 17-5) in the C50 domain includes transcendental functions, quantizers, table lookup stars, and miscellaneous nonlinear functions.

`Abs`	Output the absolute value of the input.
`ACos`	Output the inverse cosine of the input, which is in the range -1.0 to 1.0. The output, in the principle range of 0 to π, is scaled

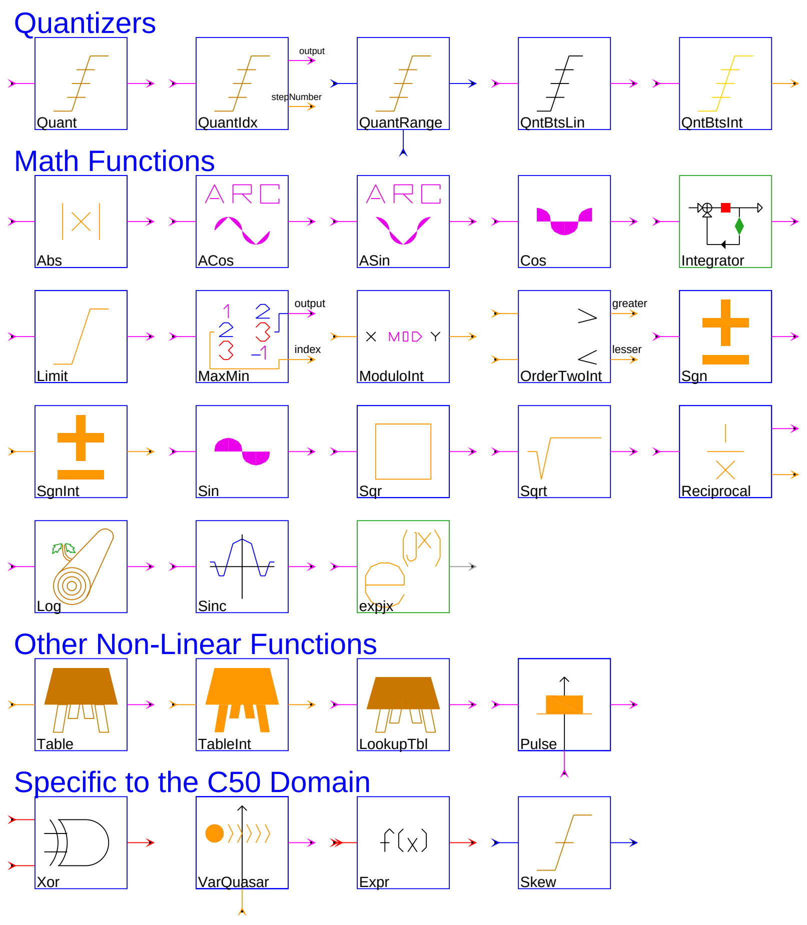


FIGURE 17-5: C50 Nonlinear Palette

	down by π.
`ASin`	Output the inverse sine of the input, which is in the range -1.0 to 1.0. The output, in the principle range of $-\frac{\pi}{2}$ to $\frac{\pi}{2}$, is scaled down by π.
`Cos`	Output the cosine, calculated the table lookup. The input range is [-1,1] scaled by π.
`expjx`	Output the complex exponential of the input.
`Intgrtr`	An integrator with leakage set by *feedbackGain*. If there is an overflow, the *onOverflow* parameter will designate a wrap around, saturate or reset operation.
`Limit`	Limits the input between the range of [*bottom*, *top*].
`Log`	Outputs the base two logarithm.
`MaxMin`	Output the maximal or minimal (*MAX*) sample out of the last *N* input samples. This can either *compareMagnitude* or take into account the sign. If *outputMagnitude* is `YES` the magnitude of the result is written to the output, otherwise the result itself is written.
`ModuloInt`	Output the remainder after dividing the integer input by the integer *modulo* parameter.
`OrderTwoInt`	Takes two inputs and outputs the greater and lesser of the two integers.
`Quant`	Quantizes the input to one of N+1 possible output *levels* using N *thresholds*.
`QuantIdx`	The star quantizes the input to one of N+1 possible output *levels* using N *thresholds*. It also outputs the index of the quantization level used.
`QuantRange`	Quantizes the input to one of N+1 possible output *levels* using N *thresholds*.
`Reciprocal`	Outputs the reciprocal to *Nf* precision in terms of a fraction and some left shifts.
`Sgn`	Outputs the sign of the input.
`SgnInt`	Outputs the sign of the integer input.
`Sin`	Outputs the sine, calculated using a table lookup. The input range is [-1,1) scaled by π.
`Sinc`	Outputs the sinc functions calculated as sin(x)/x.
`Sqrt`	Outputs the square root of the input.
`Table`	Implements a real-valued lookup table. The *values* state contains the values to output; its first element is element zero. An

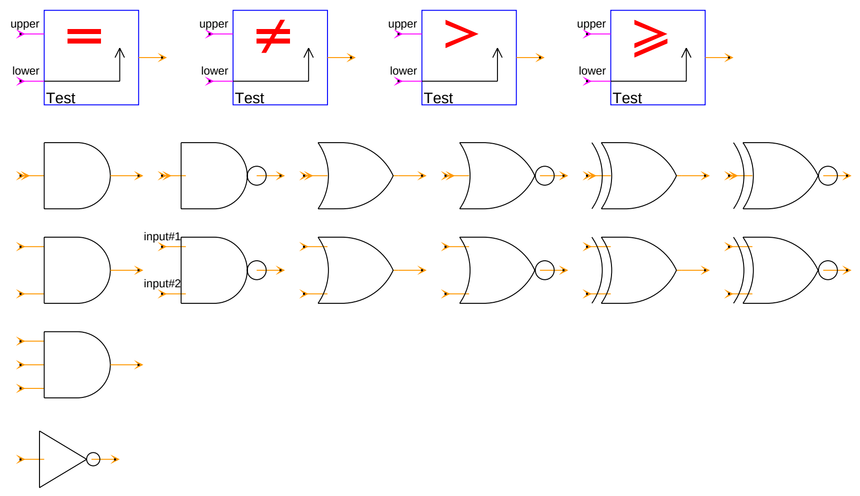


FIGURE 17-6: C50 Logic Palette

	error occurs if an out of bounds value is received.
`TableInt`	Implements an integer-valued lookup table. The *values* state contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received.
`Expr`	General expression evaluation.
`LookupTbl`	The input accesses a lookup table. The *interpolation* parameter determines the output for input values between table-entry points. If *interpolation* is “linear” the star will interpolate between table entries; if *interpolation* is set to “none”, it will use the next lowest entry.
`Pulse`	Generates a variable length pulse. A pulse begins when a non-zero trigger is received. The pulse duration varies between 1 and *maxDuration* as the control varies between [-1,1).
`QntBtsInt`	Outputs the two’s complement number given by the top *noBits* of the input (for integer output).
`QntBtsLin`	Outputs the two’s complement number given by the top *noBits* of the input, but an optional *offset* can be added to shift the output levels up or down.
`Skew`	Generic skewing star.
`Sqr`	Outputs the square of the input.
`VarQuasar`	A sequence of values(*data*) is repeated at the output with period N (integer input), zero-padding or truncating the sequence to N if necessary. A value of O for N yields an aperiodic sequence.
`Xor`	Output the bit-wise exclusive-or of the inputs.

17.2.5 Logic stars

The Logic stars are discussed below:

`Test`	(4 icons) Test to see if two inputs are equal, not equal, greater than, and greater than or equal. For less than and less than or equal, switch the order of the inputs.
`And`	(3 icons) True if all inputs are non-zero.
`Nand`	(2 icons) True if all inputs are not non-zero.
`Or`	(2 icons) True if any input is non-zero.
`Nor`	(2 icons) True if any input is zero.
`Xor`	(2 icons) True if its inputs differ in value.
`Xnor`	(2 icons) True if its inputs coincide in value.
`Not`	Logical inverter.

17.2.6 Control stars

Control stars (figure 17-7) manipulate the flow of tokens. All of these stars are polymorphic; they operate on any data type. Refer to 5.2.6 on page 5-17 for descriptions of the SDF equivalent stars: `Fork`, `DownSample`, `Commutator`, `Distributor`, `Mux`, `Repeat`, `Reverse`, and `UpSample`.

`ChopVarOffset`	This star has the same functionality as the `Chop` star except now the *offset* parameter is determined at run time through a control input.
`Cut`	On each execution, this star reads a block of *nread* samples (default 128) and writes *nwrite* of these samples (default 64),

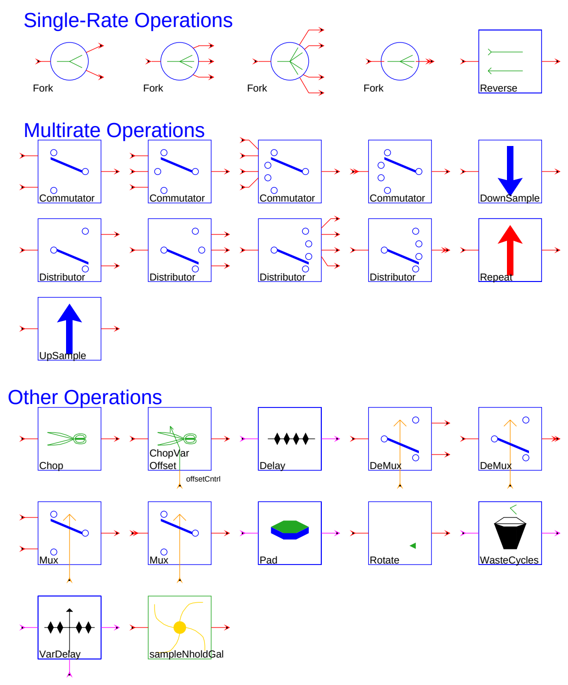


FIGURE 17-7: C50 Control Palette

	skipping the first offset samples (default 0). It is an error if *nwrite* + *offset* > *nread*. If *nwrite* > *nread*, then the output consists of overlapping windows, and hence *offset* must be negative.
`Delay`	A delay star of parameter *totalDelay* unit delays.
`Pad`	On each execution, Pad reads a block of *nread* samples and writes a block of *nwrite* samples. The first *offset* samples have value *fill*, the next *nread* output samples have values taken from the inputs, and the last *nwrite* - *nread* - *offset* samples have value *fill* again.
`Rotate`	The star reads in an input block of a certain *length* and performs a circular shift of the input. If the *rotation* is positive, the input is shifted to the left so that ouput[0] = input[*rotation*]. If the *rotation* is negative, the input is shifted to the right so that output[*rotation*] = input[0].
`sampleNholdGalaxy`	This sample-and-hold galaxy is more memory efficient than using a downsample star for the same purpose. This star is not present in Ptolemy0.6.
`VarDelay`	A variable delay that will vary between 0 and *maxDelay* as the control input varies between -1.0 and 1.0.
`WasteCycles`	Stalls the flow of data for *cyclesToWaste* number of cycles.

17.2.7 Conversion stars

The palette in figure 17-8 shows stars for format conversions from fixed point to complex fixed point.

`CxToRect`	Output the real part and imaginary part of the input of separate output ports.
`RectToCx`	Output a complex signal with real and imaginary part inputs.

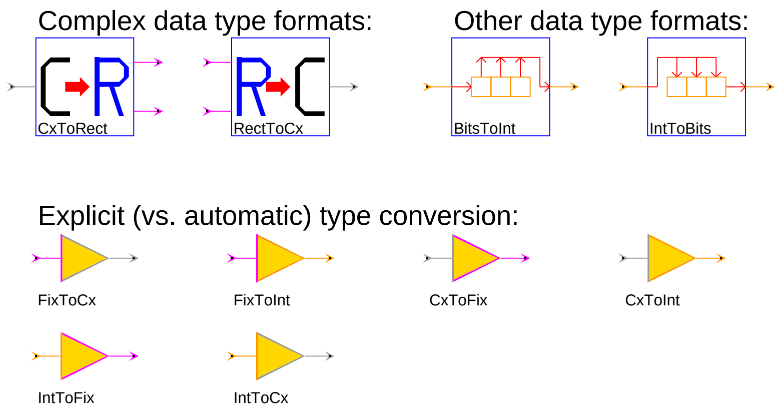


FIGURE 17-8: C50 Conversion Palette

`BitsToInt`	Convert a stream of bits to an integer.
`IntToBits`	Convert an integer into a stream of bits.
`FixToCx`	Convert fixed-point numbers to complex fixed-point numbers.
`FixToInt`	Convert fixed-point numbers to integer numbers.
`CxToFix`	Output the magnitude squared of the complex number.
`CxToInt`	Output the magnitude squared of the complex number.
`IntToFix`	Convert an integer input to a fixed point output.
`IntToCx`	Convert an integer input to a complex output.

17.2.8 Signal processing stars

The palette shown in figure 17-9 has icons for the library of signal processing functions. The filter stars follow. The `Goertzel` and `IIR` stars are identical to their SDF counterparts.

`Allpass`	An allpass filter with one pole and one zero. The location of these is given by the "polezero" input.
`Biquad`	A two-pole, two-zero IIR filter (a biquad).

$$H(z) = \frac{1 + n_1 z^{-1} + n_2 z^{-2}}{1 + d_1 z^{-1} + d_2 z^{-2}}$$

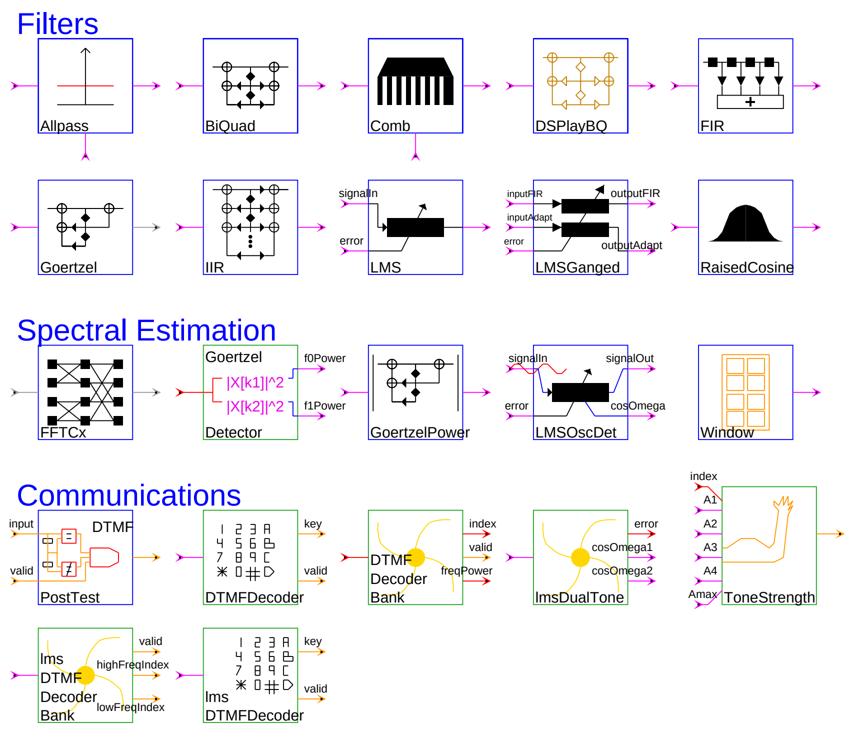


FIGURE 17-9: C50 Signal processing Palette

`Comb`	A comb filter with a one-pole lowpass filter in the delay loop.
`BiquadDSPlay`	A two-pole, two zero IIR filter (a biquad). This biquad is tailored to use the coefficients from the DSPlay filter design tool. If DSPlay gives the coefficients: A B C D E then define the parameters as follows: a=A, b=B, c=C, d=-(D+1), e = -E. This only works if a, b, c, d, and e, are in the range [-1,1). The default coefficients implement a low pass filter. $$H(z) = \frac{a + bz^{-1} + cz^{-2}}{1 - (d+1)z^{-1} - ez^{-2}}$$
`FIR`	A finite impulse response (FIR) filter. Coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear-phase, lowpass filter. To read coefficients from a file, replace the default coefficients with `< filename`, preferably specifying a complete path. Polyphase multirate filtering is not yet supported.
`LMS`	An adaptive filter using the LMS adaptation algorithm. The initial coefficients are given by the *coef* parameter. The default initial coefficients give an 8th order, linear phase lowpass filter. To read default coefficients from a file, replace the default coefficients with `< filename`, preferably specifying a complete path. This star supports decimation, but not interpolation.
`LMSGanged`	A LMS filter were the coefficients from the adaptive filter are used to run a FIR filter in parallel. The initial coefficients default to a lowpass filter of order 8.
`RaisedCos`	An FIR filter with a magnitude frequency response shaped like the standard raised cosine used in digital communications.See the `SDFRaisedCosine` star for more information.

The spectral estimation stars follow. The `GoertzelDetector, GoertzelPower`, and `LMSOscDet` are identical to their SDF counterparts.

`FFTCx`	Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter *order* (default 8) is the transform size. The parameter *direction* (default 1) is 1 for forward, -1 for the inverse FFT.
`Window`	Generate standard window functions or periodic repetitions of standard window functions. The possible functions are `Rectangle`, `Bartlett`, `Hanning`, `Hamming`, `Blackman`, `SteepBlackman`, and `Kaiser`. One period of samples is produced on each firing.

The communications stars are exactly like their SDF counterparts.

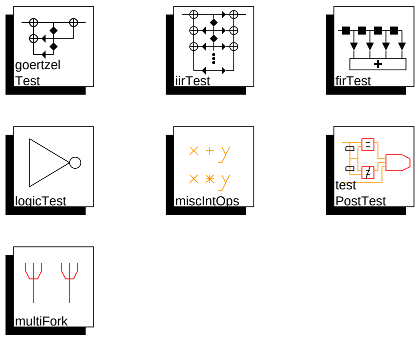


FIGURE 17-10: Basic/Test Demo Palette

17.3 An overview of C50 Demos

A set of C50 demonstration programs have been developed. The demos are meant to be run on the C50DSK board. If you do not have the required DSK tools, then you can still run the demos to see the generated code. To do this make sure that the *run* and *compile* target parameters are to `NO`. By default, the generated code is written to
`$HOME/PTOLEMY_SYSTEMS/C50` directory.

17.3.1 Basic/Test demos

The Basic/Test palette contains 7 demonstrations.

`goertzelTest`	Test the Goertzel filters for computing the discrete Fourier transform.
`firTest`	Test the finite impulse response (FIR) filters.
`iirTest`	Test the infinite impulse response (IIR) filters.
`logicTest`	Test various comparison tests and Boolean functions.
`miscIntOps`	Test integer arithmetic operations.
`multiFork`	Test the `AnyAsmFork` star. An `AnyAsmFork` star is one of a group of stars that do produce any code at compile time.
`testPostTest`	Test the `DTMFPostTest` star used in touchtone decoding.

17.3.2 DSK 320C5x demos

The DSK 320C5x demo palette contains demonstrations meant to be run on the Texas Instruments DSP Starter Kit board.

`chirp`	This system uses two integrators and a cosine to generate a chirp signal.
`DTMFCodec`	Demonstration of touchtone detection using the discrete Fourier transform implemented by using Goertzel filters.
`lms`	A noise source is connected to an eighth-order least-mean squares (LMS) adaptive filter with initial taps specifying a low-pass filter. The taps adapt to a null filter (the impulse response is

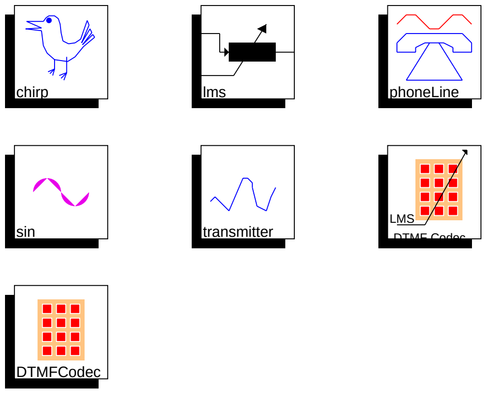


FIGURE 17-11: DSK 320C5x Palette

	an impulse) and the error signal is displayed.
`lmsDTMFCodec`	Demonstration of touchtone detection using Normalized Direct Frequency Estimation implemented by using Least-Mean Squares (LMS) adaptive filters.
`phoneLine`	A telephone channel simulator. A tone is passed through some processing which implements various distortions on a telephone channel. The parameters that are controllable are: noise, channel filter, second harmonic, third harmonic, frequency offset, phase jitter frequency, and phase jitter amplitude.
`sin`	A sine wave is generated by using two integrators in a feedback loop.
`transmitter`	A simple 4-level PAM transmitter

17.4 Targets

Three C50 targets are included in the Ptolemy distribution. To choose one of these targets, with your mouse cursor in a schematic window, execute the Edit:edit-target command (or just type “T”). You will get a list of the available `Targets` in the C50 domain. The `default-C50` target is the default value. When you click `OK`, the dialog box appears with the parameters of the target. You can edit these, or accept the defaults. The next time you run the schematic, the selected target will be used.

17.4.1 Default C50 (default-C50) target

The default target is used only for code generation. It has the following set of options:

host	(`STRING`) Default = The default is the empty string. Host machine to compile or assemble code on. All code is written to and compiled and run on the computer specified by this parameter. If a remote computer is specified here then `rsh` commands are used to place files on that computer and to invoke the compiler. You should verify that your .rhosts file is properly configured so that `rsh`

	will work.
directory	(`STRING`) Default = `$HOME/PTOLEMY_SYSTEMS/C50` This is the directory to which all generated files will be written to.
file	(`STRING`) Default = The default is the empty string. This represents the prefix for file names for all generated files.
Looping Level	Specifies if the loop scheduler should be used. Please refer to the section “default-CG” on page 13-2 for more details on this option. Refer to “Default SDF target” on page 5-65 and “The loop-SDF target” on page 5-67 for more details on loop scheduling.
display?	(`INT`) Default = `YES` If this flag is set to `YES`, then the generated code will be displayed on the screen.
compile?	This is a dummy flag since the default target only generates code.
run?	This is a dummy flag since the default target only generates code.
bMemMap	(`STRING`) Default = `768-1279` Address range for C50 Dual Access RAM blocks. C50 Instructions that operate on data run faster if the data is stored in one of the DARAM blocks. Disjoint segments of memory can be specified by separating the contiguous ranges with spaces, e.g. “768-800 1200-1279.”
uMemMap	(`STRING`) Default = `2432-6848` Data address range in the C50 Single Access RAM block. This can also specify a valid address range in external memory.
subroutines?	(`INT`) Default = `-1` Setting this parameter to N makes the target attempt to generate a subroutine instead of in-line code for a star if the number of repetitions of that star is greater than N (use N=0 to generate subroutines even for stars with just 1 repetition). Set *subroutines?* to `-1` (or any other negative integer) to disable the feature.

17.4.2 C50 Subroutine (sub-C50) target

This target is used to generate subroutines that can be called from hand-written C50 code. The options are identical to those of `default-C50` target.

17.4.3 C50 DSP Starter Kit (DSKC50) target

This target is used to generate C50 code to be run on Texas Instruments’ DSP Starter

Kit board. In addition to the regular `file.asm` generated by the other targets, this target will produce a second file (*`file`*`DSK.asm`) which is the same as the original file but with all lines truncated to 80 characters. This is done because the TI DSK assembler will give false error messages if lines in the input file exceed 80 characters. The options are identical to those of `default-C50` target with four exceptions:

compile?	If this flag is set the target will issue the command `asmc50` *`file`*`DSK.asm` where *`file`*`DSK.asm` is the name of the file containing the generated code. This should run the DSK assembler and produce a file *`file`*`DSK.dsk`. Note that `asmc50` can be a shell script that invokes the user's DSK assembler. Scripts to use the TI DSK assembler and loader in Linux are presented at the end of this section.
run?	If this flag is set the target will issue the command `loadc50` *`file`*`DSK.dsk` which should load *`file`*`DSK.dsk` to the DSK board. Note that `loadc50` can be a shell script that invokes the user's DSK loader.
bMemMap	(`STRING`) Default = `768-1270` Valid addresses on the Dual Access RAM block 1. The last 9 words in this (addresses 1271 - 1279) are reserved by the target to store configuration information for the Analog Interface Chip.
uMemMap	(`STRING`) Default = `2432-6847` Valid addresses on the Single Access RAM memory. Locations 6848 - 11263 are reserved to store the user's program and locations 2048-2431 are reserved by the TI DSK debugger kernel.

The following scripts invoke the TI DSK assembler and loader from Linux through `dosemu` (a DOS emulator). Note that before invoking the assembler and loader Ptolemy executes a cd to the *directory* target parameter. Since you need to unmount the DOS partition to run `dosemu` you can not have *directory* set to the DOS partition. One solution is to set *directory* to your home directory and set *file* to include the path to the directory where you want the file written. For example, if your home directory is `/ptuser`, the dos partition `dosemu` will use is `/dos/c` and you want the output files written to `/dos/c/dsk/src` the you could set *directory* to `/users/ptdesign` and *file* to `/dos/c/dsk/`*`filename`* where *`filename`* is the name of the output file. These scripts are also included in `$PTOLEMY/src/domains/c50.`

```
#!/bin/sh
# Version: @(#)asmc501.604/07/97
# Copyright (c) 1996-1997 The Regents of the University of California.
# All Rights Reserved.
#
# asmc50
# script to assemble files with TI's DSK assembler(dsk5a.exe)
# Uses dosemu to run dsk5a.exe.  The person running it must be root to
# mount/unmount the dos partition.
# This script was tested on a machine running linux (red-hat 3.0.3
# distribution) with dosemu-0.63.1.33 installed.
#
# Written by Luis Gutierrez.
# Converted from csh to sh by Brian L. Evans

# User’s home directory.
homedir=/root

# User’s dos partition.
dospartition=/dos/c

# The root path of DOS drive where DSK files and DOS binaries are
stored.
dosroot=c:

# The DOS directory(relative to dosroot)where the *.asm and *.dsk
files
# are stored. Replace the \ in the DOS path with \\.
dsksrc=dsk\\src

# The DOS directory(relative to dosroot) where the DSK
# executables(dsk5a.exe, dsk5l.exe) are stored.
# Replace the \ in the DOS path with \\.
dskbin=dsk

# The file used to temporarily save autoexec.emu.
autoexecsave=autoexec.bak

cd $dospartition
mv autoexec.emu $autoexecsave

# The text between the first xxxx and the second xxxx will be
# piped to unix2dos and will end up in autoexec.emu.

unix2dos > $dospartition/autoexec.emu << xxxx
path $dosroot\\$dskbin;$dosroot\\dos
cd $dosroot\\$dsksrc
dsk5a.exe $1:t
exitemu
xxxx
cd $homedir

# Unmount DOS partition to run dosemu
umount $dospartition
dos > /dev/null

# Mount DOS partition after running dosemu
mount -t msdos /dev/sda1 $dospartition

# Restore autoexec.emu
cd $dospartition
mv -f $dospartition/$autoexecsave  $dospartition/autoexec.emu
```

The following script is used to load files.

```
#!/bin/csh
# Version: @(#)loadc501.5 03/29/97
# Copyright (c) 1996-1997 The Regents of the University of California.
# All Rights Reserved.
#
# loadc50
# script to load files with TI's DSK loader(dsk5l.exe)
# Uses xdos to run dsk5l.exe.  The person running it must be root to
# mount/unmount the dos partition.
# This script was tested on a machine running linux(red-hat 3.0.3
# diistribution) with dosemu-0.63.1.33 installed.
# Written by Luis Gutierrez.
#
# Converted from csh to sh by Brian L. Evans

# User's home directory.
homedir=/root

# User's dos partition.
dospartition=/dos/c

# The root path of DOS drive where DSK files and DOS binaries are
stored.
dosroot=c:

# The DOS directory(relative to dosroot\)where the *.asm and *.dsk
files
# are stored. Replace the \ in the DOS path with \\.
dsksrc=dsk\\src

# The DOS directory(relative to dosroot) where the DSK
# executables(dsk5a.exe, dsk5l.exe) are stored.
# Replace the \ in the DOS path with \\.
dskbin=dsk

# The file used to temporarily save autoexec.emu
autoexecsave=autoexec.bak

cd $dospartition
mv autoexec.emu $autoexecsave

# The text between the first xxxx and the second xxxx will be
# piped to unix2dos and will end up in autoexec.emu.

unix2dos  > $dospartition/autoexec.emu << xxxx
path $dosroot\\$dskbin;$dosroot\\dos
cd $dosroot\\$dsksrc
dsk5l.exe  $1:t
exitemu
xxxx
cd $homedir

# Unmount DOS partition to run xdos
```

```
umount $dospartition
xdos

# After running xdos mount DOS partition
mount -t msdos /dev/sda1 $dospartition

# Restore autoexec.emu
cd $dospartition
mv -f $dospartition/$autoexecsave  $dospartition/autoexec.emu
```

Chapter 18. Creating Documentation

Authors: *Joseph T. Buck*
Christopher Hylands
Alan Kamas
Edward A. Lee

Other Contributors:
Phil Lapsley

18.1 Introduction

Since Ptolemy is by design an extensible system, the documentation must be also be extensible. This chapter explains the document formatting conventions, scripts, and sample documents that are distributed with Ptolemy. At this time, we use a combination of text processing systems for documentation. Currently, the main systems we use are FrameMaker[1] and HTML, though older version of Ptolemy used troff, and some TeX components. A version of this particular chapter is distributed as a sample document in `$PTOLEMY/doc/samples/documents.book`[2]. It is made using FrameMaker, and can be used as a template for generating new FrameMaker documents in the Ptolemy style.

For users who do not have access to FrameMaker, a compatible alternative document formatting system based on troff is also provided. Currently, all documentation for stars, galaxies, and demo programs is based on HTML. All shell scripts and Makefiles are supplied along with the documentation so that they can be modified if necessary..

18.2 Printing the manual

The simplest way to get a hard copy version of the manual is to have a double sided bound copy sent to you. You may order the documentation set from:

EECS/ERL Industrial Liaison Program Office, Software Distribution
205 Cory Hall
University of California at Berkeley
Berkeley, CA 94720
phone: (510) 643-6687
fax: (510) 643-6694
email: ilpsoftware@eecs.berkeley.edu

If you would like to print out your own copy of the documentation, you will need a postscript printer. All of the Ptolemy documentation is contained in a collection of postscript

1. FrameMaker is a registered trademark of the Frame Technology Corporation.
2. $PTOLEMY is an environment variable that is assumed to specify the installation directory for the Ptolemy system.

files. These files have the “.ps’ suffix at the end of their file names. The files are found in the Ptolemy distribution as follows:

`$PTOLEMY/doc` contains most of the documentation. Within `$PTOLEMY/doc`, the directory `users_man` contains the Ptolemy user’s manual. The directory `prog_man` contains the programmer’s manual. The `bin` directory contains scripts for building and printing troff based documentation. The `headers` directory contains troff header files that are needed by the scripts in the `bin` directory. The `main` directory contains the makefiles needed to print out troff based documents. Finally, the directory `$PTOLEMY/doc/samples` contains sample documents and templates to follow if you are planning to add new documentation.

Domains and their stars are documented where their source code resides. For instance, the documentation for the SDF domain is in `$PTOLEMY/src/domains/sdf/stars`. The format for this documentation, and methods for printing it are found in the section titled “Using HTML to document stars” on page 18-5.

Each of the documentation directories mentioned above may have a `README` file that explains which postscript files are which and explains how to print out the files. In any event, you can print the section you are interested in by going to the documentation directory and then printing out the postscript files found there.

18.3 Using FrameMaker

In the directory `$PTOLEMY/doc/samples`, the following files can be found:

`title.doc`:	A sample cover page.
`documents.doc`:	A sample document (very similar to this chapter).
`documentsTOC.doc`:	The table of contents for the document.
`documents.book`:	A book file that unites the above three documents.

These documents can be used as models or templates for creating new documents that are to be inserted in *The Almagest* or are to stand alone. Use `documents.doc` as a template for most applications. It defines the paragraph and character styles visible in this chapter.

By convention, except for the sample document, we do not distribute the FrameMaker files for the entire *Almagest*. Instead, we distribute the PostScript[1] code produced by Frame. The makefiles used to print manual, therefore, simply assume that the PostScript files are up to date. It is up to you to ensure this. You must also ensure that the index files corresponding to the PostScript code are up to date. The section below explains how to generate these.

18.3.1 Index Entries

We use FrameMaker to generate the indexes for each manual. Different index markers are used to denote different uses of the term bein indexed. For example, the definition of a star gets a different FrameMaker marker than a simple reference to the star. In the index file the page number of the definition will be in bold, the page number of the reference will be in a regular font.

To use the Ptolemy group markers, the following X resources should be modified in

1. PostScript is a registered trademark of Adobe Systems Inc.

the file `$FMHOME/fminit/usenglish/Maker.us`:

```
Maker*marker.10: IndexReference
Maker*marker.11: IndexExamplle
Maker*marker.12: IndexDefinition
Maker*marker.13: IndexStarRef
Maker*marker.14: IndexStarEx
Maker*marker.15: IndexStarDef
Maker*marker.24: HTMLStart
Maker*marker.25: HTMLEnd
```

These resources cause the named index markers to appear in the list of markers.

To make an index entry in a FrameMaker document, select the text you wish to appear in the index, and select the FrameMaker command Special-Marker (`Esc-s-m`). Then choose one of the above six types of index entries, using the following guidelines:

`IndexReference`:	Generic index entry
`IndexExample`:	An example of the usage of a particular feature.
`IndexDefinition`:	The definition of a term.
`IndexStarRef`:	A generic reference to a star.
`IndexStarEx`:	An example of the usage of a star. For example if the text that describes the SDF `butterfly` demo would have a index entry that looks like: `butterfly (SDF demo)`.
`IndexStarDef`:	The definition of a star. This entry is normally automatically generated when a star is compiled, so you will probably not encounter any occasion to use it directly. The text that defines the SDF `Ramp` star would have the marker text: `Ramp (SDF block)`

Avoid index entries beginning with very generic words in the Ptolemy vocabulary, like Ptolemy, `star`, `galaxy`, or `domain`. Of course, if you are writing some explanation of these basic terms, then an index entry is appropriate. Before entering index entries for a star, look in the documentation for similar stars to get an idea of the subject terms that have already been used and might be related. Be sure to follow the same capitalization rules as the existing index entries (i.e. `Ramp (SDF block)`, not `Ramp (SDF Block)`).

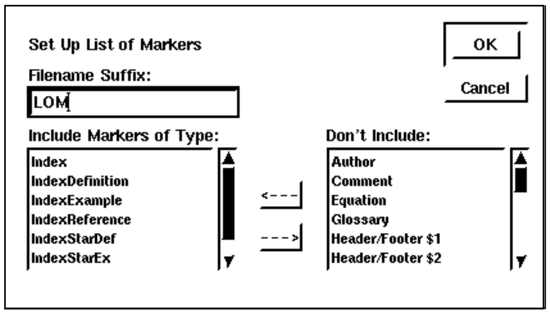
Currently we use Quadralay's WebWorks to convert Framemaker documents to html. You can use `HTMLStart` to indicate text that should be present in the html output as a link. Use a `HTMLEnd` on the end of the text of the text that represents the link. The text of the first `HTMLStart` contains the filename or URL:

```
http://ptolemy.eecs.berkeley.edu
```

The `HTMLEnd` is placed at the end of the text you want underlined in the html version.Here is a sample link to `http://ptolemy.eecs.berkeley.edu`.

The markers are not printed when the FrameMaker document is printed. WebMaker converts the text delimited by the markers into HTML hypertext links.

When you print your document, you should generate the index file that will be used to print the overall index. To do this, select File/Generate, and within the ensuing dialog box, select List/List of Markers. In the dialog box that results from this, be sure all of the above index markers are included, and then accept the default filename suffix “LOM”. The dialog box should look like this:



Filename Suffix: LOM

Include Markers of Type:	Don’t Include:
Index	Author
IndexDefinition	Comment
IndexExample	Equation
IndexReference	Glossary
IndexStarDef	Header/Footer $1
IndexStarEx	Header/Footer $2

When you click OK, you will get a new file with a list of markers in a format acceptable to the Ptolemy index generation software. This file should be saved in “Text only” form. By convention, we name the index file using the document name with the suffix “.index”.

18.3.2 Special fonts and displays

- By convention, Ptolemy documentation uses a special font for C++ class names. In the FrameMaker template, the corresponding format is named `Class`.

```
For a display entirely set in this font,
use the "Commands" paragraph format, as shown here.
You can use "meta-Return" to force carriage returns where
you want them without getting a new paragraph.
```

- Note that if you use the Commands paragraph format, you should be sure to change any slanted double quotes “ ” to straight quotes `"`, by typing Control-Shift-`"`. In general, commands don’t have slanted double quotes, hence the need to convert them to straight quotes.
- Star and Target parameters, such as the SDF *loopScheduler* target parameter, are always in italic, use the Emphasis font format.
- The names of stars and demos should be in the `Class` format.
- If a string is to be taken literally, it should be in the `ProgramCode` character format. An example would be that the default of the *loopScheduler* target parameter is `1`. Strings such as `YES`, `NO`, `TRUE` and `FALSE`, that are used as values to parameters should also be in the `ProgramCode` format.
- The first line of a dialog with a computer should use the `Commands` paragraph format. Each successive line should use the `CommandsCont` paragraph format. The text the computer would print should be in the `ProgramCode` character format, the text the user would type should be in the **`ProgramUser`** character format.
- If a string is only an example and to be substituted with a proper value, it should be in *`ProgramVariable`* character format.
- To include images of palettes in documents, see “Capturing a screen image” on page 2-41.

18.4 Using HTML to document stars

Stars are currently documented in HTML. The `ptlang` program processes a `.pl` file and produces `.cc`, `.h` and `.html` files in the same directory as the `.pl` file. The contents of the `htmldoc` section of the `.pl` file end up in the `.html` file. See the Programmer's Manual for details.

The Tycho on-line documentation includes a style guide in `$PTOLEMY/tycho/doc/documentation.html`.

Chapter 19. Vem — The Graphical Editor for Oct

Authors:	*David Harrison*
	Rick Spickelmier
Other Contributors:	*Bill Bush*
	Andrea Cassotto
	Christopher Hylands
	Edward A. Lee

19.1 Terminology

`Vem` is an interactive graphical editor for the `oct` design database. It was written by David Harrison and Rick Spickelmier in the CAD group at UC Berkeley. It has been extended by Andrea Cassotto and Bill Bush. An introduction to the terminology used in the system is given in "The oct design database and its editor, vem" on page 2-21. In this chapter, we give more detailed information about `vem`. Most users will not need this much detail; chapter 2 will be enough.

Most of this chapter is extracted from standard documentation for the `octtools` distribution. No `oct` documentation is included. See "Customizing Vem" on page 19-23 for other resources.

The fundamental `oct` objects that we edit with `vem` are called *facets*. Facets are specified by three names separated by colons. This is usually written as "cell:view:facet". The first component is the cell name; it is used to name the design. Note that cell name may not contain any spaces. In `pigi`, the second component, called the view name, will always be "schematic".[1] Third, is the "facet" component, which can either be "contents" or "interface". The former specifies a block diagram, while the latter defines an icon. This usage of the term "facet" is different from our previous usage. Thus, "facet" can mean either the `oct` object called a facet, or "contents" vs. "interface". The intended meaning is usually clear from context. In commands that depend on the facet (in the latter sense), if you do not specify it, `vem` assumes that you mean the contents facet. Thus, "wave:schematic" refers to the facet with cell name "wave", view name "schematic", and facet name "contents".

`Vem` was originally written with VLSI designs in mind. Ptolemy is an attached tool, invoked via a program called `pigiRpc`. `Vem` was originally intended for IC design. As such `vem` provides standard graphics editing capabilities for physical (mask-level), symbolic, and schematic designs. Ptolemy uses schematic capabilities for applications and physical capabilities for icons.

`Vem` may be started by simply typing `vem`, but this will not start Ptolemy. To start the

1. Other oct applications use other views such as "symbolic" or "physical."

Ptolemy interactive graphical interface, simply execute the command pigi in $PTOLEMY/bin. This invokes a shell script that starts vem and the associated pigiRpc process. Vem is started in general with the following command line options:

```
vem [-F cell[:view[:facet]]] [-G WxH+X+Y] [-R [host,]path] \
     [-display host:display] [name=value ...]
```

For example, the following script could be used to start Ptolemy on a Sun 4 workstation:

```
xrdb -m $PTOLEMY/lib/pigiXRes9
vem -G 600x150+0+0 -F init.pal:schematic \
     -G 600x300+0+170 -R $PTOLEMY/bin.sun4/pigiRpc
```

The first line merges the X Windows resources defined in $PTOLEMY/lib/pigiXRes9. The next line starts vem and the associated pigiRpc process. The pigi script is simply a more elaborate version of this that ensures the existence of the init.pal facet and sets up the user's environment.

Vem looks at the value of the DISPLAY environment variable to determine what host and display to use for output. Vem and pigi may be run in the background without affecting the program operation.

The -F, -G, and -R command line options allow a user to specify a start-up window configuration for vem. These three options are considered triplets that specify the initial cell, position and size, and remote application respectively for a window. There is no limit to the number of triplets that may be specified. The -F flag marks the start of each new triplet. The corresponding -G and -R flags after the -F flag are optional. If the -G flag is omitted, vem will not specify a location for the window and most window managers will interactively prompt for the window location. If the -R flag is omitted, no remote application will be started in the window. The -F flag can be omitted from the first triplet. In this case, the -G and -R flags apply to the console window. For example, the pigi script above starts vem with its console window at (0,0) with a size of 600 by 150, and one window looking at the cell "init.pal:schematic" at (0,170) with a size of 600 by 300, running the pigiRpc remote application.

Vem is a highly customizable editor. Nearly all of the colors, font styles and fill patterns vem uses can be changed by the user. Normally, these parameters are read from the user's X resources (which are usually loaded when X is started from a file named ~/.Xdefaults, or something similar). However, one can set certain parameters on the command line using the = (equal) command line option. A list of all configurable parameters can be found in the document "Customizing Vem," which is distributed with the standard octtools distribution. This document can also be found as $PTOLEMY/src/octtools/vem/doc/Vemcustom.ps.

The console window echoes user input and outputs various help and status messages. After starting, the console will display a prompt and wait for input. Ptolemy users rarely need to use this window, and eventually, it will be eliminated.

If the init.pal facet does not exist in the directory in which pigi is started, then it will be created. A blank facet will appear. Convention in Ptolemy dictates that this facet should be used to store icons representing complete applications, or universes, that are defined in the directory. If such icons already exist in the init.pal facet, the applications can be examined using the `pigi` “look-inside” command.

New windows can be created using *open-facet* command (see table 2-2 on page 7). It is also possible to open a window from the `vem` console using the *open-window* command, but this new window will not be attached to `pigiRpc` (see the command reference below). This means that you will not be able to issue the `pigi` commands in table 2-2 from these windows.

Each window has exactly one associated cell. Mouse action with the cursor positioned inside a window cause operations to occur to the associated cell. Any number of windows can be created with the same or different associated cells. More than one window may have the same associated cell. In this case, all of the windows are attached to the *same* cell. Thus, a change to one of the windows may cause updates to other windows that look at the same cell.

`Vem` assumes a three-button mouse. The left button is used for entry of graphics information. The middle button is used for the primary menu of commands. The right button is used to modify graphics information entered using the left button.

Commands to `vem` are specified in post-fix form. The user builds an argument list first and then selects a command. Commands can be selected in three ways: pop-up menus, single keystrokes, or by typing in the command name. Pressing and releasing the middle button in a graphics window causes a `vem` menu to appear. The user can use the mouse to riffle through the options until the desired choice is highlighted. The commands are summarized in table 2-3 on page 11. Pressing and releasing the mouse button activates the selected command. Pressing and releasing the mouse outside the menu cancels the selection. Normally, pressing and releasing the middle button causes a `vem` menu to appear. Holding the shift key and clicking the middle button causes the `pigi` menu to appear. Both menus are useful.

A number of common commands can be selected via a single keystroke. Key bindings for various commands are shown next to the corresponding entry in the `vem` menu, are listed in the command reference below, and can be queried interactively using the *bindings* command. Typing a colon (`:`) allows the user to type in the command name (or a user defined alias) in the console window. The standard line editing keys can be used while typing the command name. This interface supports automatic command completion. Typing a tab will complete the command if it is unique or offer a list of alternatives if it is not unique. The command is selected by typing a carriage return.

There are five types of input to `vem`: points, boxes, lines, text, and objects. Points are entered by pressing and releasing the left button of the mouse. Boxes are entered by pressing, dragging, and releasing the left button. Lines are entered by pressing, dragging and releasing the left button over a previously created point or line. Text is entered by typing the text enclosed in double quotes. If entering a filename, typing a `Tab` character will cause `vem` to try to complete the name if it is unique or offer a list of alternatives if it is not unique. Objects are entered using the *select-objects* and *unselect-objects* commands. The last item on an argument list can be deleted using the standard character for delete. The last group of items can be deleted using the word erase character `Control-W`. The entire argument list can be deleted using the standard kill-line character (usually `Control-U`).

Once entered, graphics arguments (points, boxes, and lines) can be modified in various ways. For all arguments, a point can be moved by moving the cursor over the point and pressing, dragging, and releasing the right mouse button. New points can be added to a group of lines by moving over a point in the segment, depressing, dragging, and releasing the left mouse button. This will insert a new point after the point. It is also possible to interactively move, rotate, and mirror object arguments (selected items). See the description of the *transform* command in the command reference below.

This version of `vem` supports three basic editing styles: physical, symbolic, and schematic. Physical editing involves the entry and editing of basic geometry and the creation of interface terminals. This style is used in `pigi` to build icons. Symbolic editing involves the placement of instances of leaf cells and the interconnection of these instances. `Pigi` does not use symbolic editing. Schematic editing is an extension of symbolic where the primitive cells are schematic symbols and wire width is insignificant. Schematic cells use used by `pigi` to represent block diagrams.

When `vem` opens a new cell directly (i.e. not via `pigiRpc`), a dialog will appear asking for three property values: TECHNOLOGY, VIEWTYPE, and EDITSTYLE. The TECHNOLOGY and VIEWTYPE properties determine the location of the technology facet, which specifies the colors and layers in the display. A standard technology facet has been designed for Ptolemy, so the defaults that appear are almost always acceptable. Layer display and design rule information is read from this facet. The EDITSTYLE property is used by `vem` to determine the set of commands available for editing the cell. Currently, the legal editing styles are PHYSICAL, SYMBOLIC, or SCHEMATIC.

19.2 Using Dialog Boxes

Some commands require information that cannot be expressed easily using post-fix notation. Examples include destructive commands that require an explicit confirmation and commands that require complex non-graphic information. `Vem` uses *dialog* boxes based on the MIT Athena widgets to handle these situations. Dialog boxes are windows that resemble business forms. These windows contain labeled fields for entering text, changing numerical values, and selecting options. This section describes how to use dialog boxes.

All dialog boxes in `vem` have the same form. An example is shown on page 2-14, and also in figures 19-1 on page 12 and 19-2 on page 14. At the top of all dialogs is a one line title indicating the purpose for the dialog. The middle of the dialog (known as the body) contains fields for displaying and editing information of various kinds. At the bottom of the dialog are a number of *control* buttons. Each control button represents a command. The arguments to the command are the values of the fields displayed in the body. Thus, operating a dialog consists of editing or changing fields in the body and then selecting a command by activating a control button. Six kinds of fields may appear in the body of a `vem` dialog: editable text, non-editable text, enumerated value, numerical value, exclusive lists, and non-exclusive lists. A description of each field type is given in the paragraphs that follow.

Editable text fields are used to enter and edit text. Visually, an edit text field consists of a box containing a caret cursor, an optional scrollbar, and a label to the left of the box indicating the purpose for the field. Only one editable text field is active in any one dialog. The active editable text field has a dark border. Typing text with the mouse positioned anywhere in the dialog inserts text into the active editable text field. Most of the basic emacs editing commands can be used to modify the text in the field, as shown in table 19-1.

The insert position in the field may also be changed by pressing the left mouse button when the mouse cursor is over the desired position. Any editable text field can be made active by clicking the left mouse button inside the editable area. Alternatively, one can use the `Tab` key to make the next text field active and `Meta-Tab` to make the previous field active. Editable text fields that display large amounts of text have a scrollbar to the left of the text area. Pressing the left and right mouse buttons when the mouse cursor is in a scrollbar will scroll the text down and up respectively in proportion to the distance between the mouse cursor and the top of the scrollbar. As an example, pressing the left mouse button near the bottom of the scrollbar will scroll down the text almost one screen. Pressing and releasing the middle mouse button scrolls the text to a relative position based on how far the mouse cursor is from the top of the scrollbar. Holding down the middle mouse button will interactively scroll through the text.

Non-editable text fields are used to display text messages. They consist of a box containing text and an optional scrollbar. The scrollbar operates just like those used in editable text fields.

Enumerated value fields are used to specify one value out of a small list of values. They consist of a value displayed inside a box and a descriptive label to the left of the value. The border of the value highlights as the mouse cursor moves over it. Depressing and holding the left mouse button inside the value box causes a menu to appear that displays all possible values. The choices will highlight as the mouse cursor moves over them. To select a new value, release the mouse button when the desired choice is highlighted. The new value will appear in the value box. One can leave the value unchanged by releasing the mouse button outside the menu boundary.

A numerical value field is used to specify a magnitude between a predetermined minimum and maximum. Visually, it consists of a box containing a numerical value, a horizontal scrollbar to the right of the box for changing the value, and a label to the left describing the

Key	Description
delete, control-h	Delete previous character
control-a	Move to beginning of line
control-b	Move backward one character
control-d	Delete next character
meta-d	Delete next word
control-e	Move to end of line
control-f	Move forward one character
meta-i	Include a file
control-k	Kill (delete) to end of line
control-n	Next line
control-p	Previous line
control-s	Search forward
control-v	Next page
meta-v	Previous page
control-y	Yank deleted text

TABLE 19-1: Emacs-style text editing commands supported in vem dialog boxes.

value. The magnitude of the value is changed by operating the scrollbar. Pressing the left and right buttons in the scrollbar decrement and increment the value by one unit. Pressing the middle button changes the value based on the distance between the mouse cursor and the left edge of the scroll bar. The middle button may be pressed and held to interactively modify the value. Most people use the middle button to set the value roughly then use the left and right buttons to make the value precise.

Exclusive lists are used to choose one possible value out of a (possibly quite large) list of values. These values are displayed in a box with a scrollbar on the left edge of the box. Each value consists of a button box on the left and a descriptive label to the right. As the mouse moves over a button box, it will highlight to indicate it can be activated. The button box of the selected item will appear dark while all others will remain light. If there are too many values to display in the box at one time, the scrollbar can be used to scroll through the possible values. The scrollbar operates in the same way as described for editable text fields.

Non-exclusive lists are used to choose zero or more possible values from a (possibly quite large) list of values (see figure 19-2 on page 14). A non-exclusive list resembles an exclusive list both in appearance and operation. However, unlike an exclusive list, one can choose any number of items in a non-exclusive list. Visually, the two lists are distinguished by the appearance of the button boxes. Exclusive button boxes resemble radio buttons. Non-exclusive button boxes resemble check marks. Pressing the left button in a non-exclusive button box causes the value to toggle (i.e., if it was selected it becomes unselected, if it was unselected it becomes selected).

Control buttons cause the dialog to carry out some operation. They consist of a text label surrounded by a box. Control buttons are activated in one of two ways: pressing and releasing the left mouse button when the mouse cursor is positioned inside the button boundary, and through keystrokes. Not all control buttons can be activated using keystrokes. Those that can be activated in this fashion display the key in parentheses under the button label. Although there are exceptions, most dialogs support the keyboard accelerators given in table 19-2.

Dialogs may be both *moded* and *unmoded*. Moded dialogs are those requiring a response before processing can proceed. `Vem` uses these kinds of dialogs to ask for confirmation before proceeding. On the other hand, unmoded dialogs remain active until explicitly dismissed by the user. Other commands may be invoked freely while unmoded dialogs are visible. Most non-confirmation dialogs in `vem` are unmoded.

Key	Action
<return>, <meta-return>, <F1>	OK
<delete>, <meta-delete>, <F2>	Cancel
<Help>, <F3>	Help
<F4>	All
<F5>	Clear

TABLE 19-2: Keyboard accelerators supported by most vem dialog boxes.

19.3 General Commands

Below is a reference for all `vem` commands. This section outlines general commands available for editing all types of cells. Section 19.4 discusses options, section 19.5 describes the general selection mechanism. and section 19.6 describes property and bag editing features. The next three sections describe editing commands for physical, symbolic, and schematic cells respectively. The summary includes the name of the command as it appears on a menu, if it appears in the menu. The command name can be typed in as well. Place the mouse in the window where you wish to execute the command, enter the command arguments (points, objects, etc.), type a colon (`:`), and type the command name. The TAB character will automatically complete command names. The phrase <no-name> implies it has no default menu or command name binding.

The list below also shows the default keyboard binding for each command, if it has one, and the syntax of the argument list passed to it. The symbol <*> implies the command has no default key binding. In general, the commands used most often have key and menu bindings. Less often used commands may have only command name bindings. See table 2-3 on page 11 for a concise summary.

Some `vem` commands are not documented here because they are dangerous or conflict with the objectives of `pigi`. Those commands will not appear in the `vem` menu, and have no key binding, although all are still available by typing them in. Adventurous users may wish to consult the standard `octtools` documentation before using them.

```
<no-name>       Delete or Control-H
                             Any Argument List
```

This command deletes the last item of the last argument on the argument list. Thus, if the last argument is 10 boxes, it will delete the last box entered and the argument list will be modified to contain 9 boxes.

```
<no-name>       Control-W  Any Argument List
```

This command is similar to the one above, but deletes the entire last argument on the argument list. Thus, if the last argument is 5 lines, it will delete all 5 lines and leave the remaining arguments unchanged.

```
<no-name>       Control-X or Control-U
                             Any Argument List
```

This command erases the entire argument list allowing the user to start over.

```
bindings                     Saves Arguments
```

This command asks the user for a command and displays all of its current key, menu, and alias bindings. The command will display a prompt (`vem` bindings>) and the user can specify a command using any of the four means of normally specifying commands (via menu, single keystroke, type-in, or last command). The command also outputs a one line description of the command for help purposes.

```
close-window    Control-D  No Arguments
```

The *close-window* command closes the window the cursor was in when the command was invoked. This DOES NOT flush the contents of the window to disk. Even after all windows looking at a facet are closed, the contents are not saved on disk. This must be done using the *save-window* or *save-all* commands.

```
deep-reread      <*>         [objs]
```

The *deep-reread* command is a specialized form of the re-read command. With no arguments, it re-reads a facet and all of master facets of its subcells (instances). Both the contents and interface facets of the instances are re-read. If a set of objects is specified, the command re-reads the master cells of the instances in the object set. Only the master cells of the instances are re-read; cells are not re-read recursively when using this form.

```
interrupt        ^C          No Arguments
```

This routine interrupts (deactivates) the window containing the cursor. No drawing will be done in the window until a full redraw requested by the user (using pan, zoom, or redraw-window) is done. The key binding for this command can also be used while a window is drawing to immediately stop drawing in that window.

```
kill-buffer      <*>         "cell view {facet} {version}"
```

The *kill-buffer* command flushes a facet out of memory *without* saving its contents. If the string specification of the facet is missing, the facet is determined by the window containing the cursor. *All* windows looking at this facet are destroyed. There are no key or menu bindings for the command and it will ask the user for confirmation before carrying out the command.

```
log-bindings     <*>         Saves Arguments
```

The *log-bindings* command writes out a description of all type-in, menu, and key bindings for all commands in the editing style of the window containing the cursor. This description is written to the log file for the session.

```
open-window      o           [box] or
                             "cell:{view:{facet:{version}}}"
```

The *open-window* command is primary way to create new graphics windows in `vem`. It takes a string specifying the cell to open. When specifying the cell portion of the name, typing a TAB will attempt to complete the string as a file or offer alternatives if the name is not unique. If this string is absent, it will duplicate the window containing the cursor. Normally, the extent of the duplicated window is the same as the parent window. However, if the user specifies a box, the duplicated window will be zoomed to that extent (see zoom-in). The string specifying the cell contains four fields. The last three are optional and default to “physical”, “contents”, and the null string respectively. It is possible to specify your own defaults for these fields. Newly created windows are always zoomed to contain all geometry in the cell. If the cell does not exist, it will be created. When creating new cells, `vem` prompts the user for required cell properties. See the introduction for details. Most of the time, the defaults presented in this dialog are acceptable and activating the *Ok* button is sufficient to proceed.

```
palette          P           {"palette-name"}
```

The *palette* command opens a new window onto a previously created facet which contains standard layers or instances for a given technology. This window can be used to select layers for creating geometry or instances for instantiation. The command takes one argument: the name of the palette. If omitted, it defaults to “layer”.

Palette cells are found using the function tapGetPalette (see tap(3)). For all standard technologies and viewtypes, there is a “layer” palette. In the symbolic editing style, there are also “mosfet” and “connector” palettes which display mosfets and connectors respectively. In the schematic editing style, there are “device” and “gate” palettes which contain device level and gate level schematic primitives. New palettes can be added easily. See “Customizing Vem” for details.

```
pan             p          [Any Arguments] [point]
```

The *pan* command centers the window containing the cursor around the last point on the argument list. The window will be redrawn so that the argument list point is now the center of the window. The point need not be in the same window as the cursor. Thus, a user can point in a window showing a large portion of a cell and invoke the command in a more detailed window for a magnifying glass effect. If the point is omitted, the command assumes the cursor position is also the desired center point. This is the fast way to pan in a single window.

```
pop-context     )          No Arguments
```

This command pops off an input context from the context stack and replaces the current context with that context. See the *push-context* command for details.

```
push-context    (          No Arguments
```

This command pushes the current argument list context onto the context stack and gives the user a new context. This can be used to do other commands while preserving entered arguments. Note that the current arguments remain displayed. The old context can be restored using the *pop-context* command. Four context levels are supported in the current version of `vem`.

```
push-master     <*>        {"facet"}
```

The *push-master* command opens a new window on the master of the instance under the mouse cursor. This command can be used all editing styles. If a facet name is supplied, the command will use that facet name instead of “contents”. In Ptolemy, this command is rarely needed. The *edit-icon* command accomplishes the same objective.

```
recover-facet   <*>        No Arguments
```

Unless directed otherwise, `vem` saves all cells occasionally in case of a system crash or some other unforeseen disaster. Whenever a new cell is opened, `vem` checks to see if the last automatically saved version is more recent than the user saved version. If the automatically saved version is more recent, a warning is produced and the user version is loaded. One can use the *recover-facet* command to replace the cell with the more recent automatically saved version. The command displays a dialog containing a list of all of the saved versions. Generally, there are two possible alternative versions for a cell. The *autosave* version is written by `vem` automatically after a certain number of changes to the cell. The *crashsave* version is written when `vem` detects a serious error. Note that the crashsave version may itself be corrupt since a serious error occurred just before it was written. This command is destructive: it replaces the cell with the selected alternate. One can use the *Cancel* button to abort the recovery. Before using the *recover-facet* command, it is often useful to view the alternate cells. This can be done by specifying the version (either “autosave” or “crashsave”) as the last field in the cell specification to *open-window.*

```
re-read          <*>         No Arguments
```

The *re-read* command flushes the facet associated with the window containing the cursor out of memory and reads it back in from the disk. This can be used to see changes to a cell that were done outside `vem` or to revert back to the cell before changes were made. This is a dangerous command and `vem` asks for confirmation before proceeding.

```
redraw-window    Control-L   [box]
```

This command redraws the contents of the window containing the cursor. It does not effect the argument list (i.e. it can be done regardless of the argument list contents). If a box is provided, only the portion of the window in the box is redrawn. If the window is interrupted, this command will reactivate it. However, if the box form is used, `vem` will only draw the specified area and leave the window deactivated. This can be used to selectively draw portions of a deactivated window.

```
same-scale       =           [Any Arguments] [point]
```

This command changes the scale of the window containing the mouse to the same scale as the window containing the last point on the argument list. It is commonly used to compare the sizes of two facets.

```
save-window      S           [Any Arguments]
```

This command saves the contents of the facet associated with the window where the command was invoked. It asks for confirmation and does not effect the argument list.

```
set-path-width   w           [box] [line] [point] or
           ["size string"]
```

This command sets the current path width for a given layer on a window-by-window basis. It takes one argument which may be points, lines, boxes, or text. For points and lines, the argument length must be two. The path width is set to the maximum Manhattan distance between the points. For boxes, only one box is allowed and the new width is the larger of the two dimensions. For text, the string should contain the path width in lambda. If no width is specified, the path width will be set to the minimum layer width as specified in the technology.

The layer for the path width command is determined by looking at the object under the cursor. If there are objects on more than one layer under the cursor, a dialog will be presented and the user should choose one of the listed layers and press “OK” to continue.

```
show-all         f           No Arguments
```

The *show-all* command causes `vem` to zoom the window containing the cursor so that all of the cell is displayed in the window. The key binding is an abbreviation for “full”. It does not effect the argument list.

```
switch-facet     <*>         ["cell view {facet {version}}"]
                       or    [point]
```

This command replaces the facet in the window containing the mouse with a different facet. The first form replaces the window’s facet with the named facet. The second form replaces the window’s facet with the facet of the window containing the point.

```
toggle-grid      g          No Arguments
```

The *toggle-grid* command toggles the visibility of the grid in the window containing the cursor.

```
version          V          No Arguments
```

This command outputs the current version of `vem` to the console window.

```
where            ?          No Arguments
```

The *where* command can be used to find out the position of the cursor in terms of `oct` units. It also displays a textual representation of the objects under the cursor. The command can be issued while building an argument list without effecting the list. Alternatively, if the argument list includes an object set, the where command will print textual descriptions of the selected items.

```
write-window     W          "cell:view" [any arguments]
```

The *write-window* command saves the contents of the cell associated with the window where the command was invoked under another name. This alternate name is specified by the “cell:view” argument.

```
zoom-in          z          [Any Arguments] [box]
```

The *zoom-in* command zooms the window containing the mouse to the extent indicated by the last box on the argument list. The box and the zoom window must be in the same facet. However, the extent may be in a different window from the mouse which can be used to achieve a magnifying glass effect. If the box is not provided, it zooms in the window containing the mouse by a factor of two. If provided, the command removes the box from the argument list but leaves other arguments untouched.

```
zoom-out         Z          [Any Arguments] [box]
```

The *zoom-out* command is the opposite of zoom-in. This command zooms the window containing the mouse out far enough so that the OLD contents of the window are contained in the extent of the box provided on the argument list. Thus, smaller boxes zoom out farther than larger ones. If the box is omitted, the window containing the mouse is zoomed out by a factor of two.

19.4 Options

The options commands below all post form windows which allow a user to change display parameters interactively. The default values for these parameters can be changed in your `~/.Xdefaults` file. See the separate document “Customizing Vem” for details.

All of the options dialogs below are *unmoded*. This means that the user can do other things while the dialog is posted. They will not go away until explicitly closed by the user. Details on the operation of dialog boxes can be found in “Using Dialog Boxes” on page 19-4.

```
window-options   <*>        No Arguments
```

This command posts the dialog in figure 19-1, which presents a number of window related options each of which can be modified by the user on a window-by-window basis. Two kinds of options are presented in this dialog: flag options and value options. Flag options have a check box on the left of the option and can be either on or off. Pressing the left mouse button in the check box toggles the value of the option. Value options display a numerical value in a box with a descriptive label to the left and a scrollbar to the right for changing the value. At the bottom of the dialog are four control buttons: *Ok, Dismiss, Apply,* and *Help.* Pressing the left mouse button inside the *Ok* button saves any options you may have changed and closes the window. The *Dismiss* button does not save any options and closes the window. The *Apply* button saves any changed options but does not close the window. The *Help* button will open a window containing a brief description of the dialog. Each of the options and their meanings are given below:

“Visible Grid”

If set, a grid will be shown in the window.

“Dotted Grid”

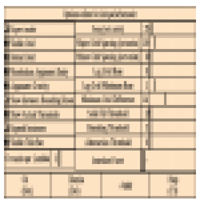
If the grid is visible and this option is set, the grid will be drawn as dots rather than lines.

“Manhattan Argument Entry”

If this option is set, entering line arguments and dragging option sets will be restricted to Manhattan angles. This option is set by default in the symbolic and schematic editing styles.

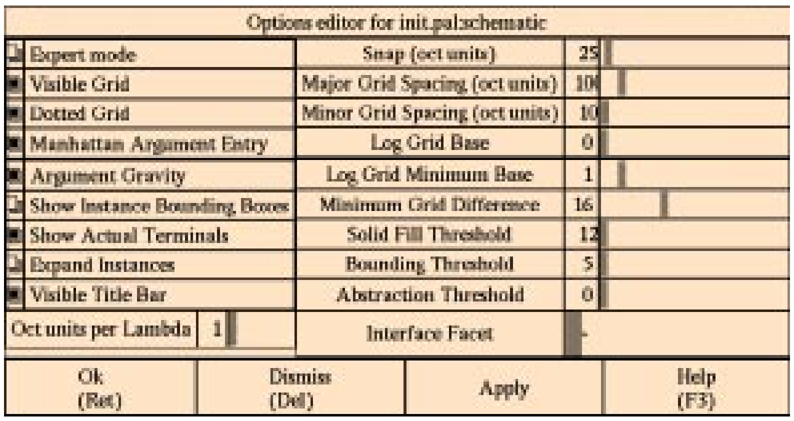
“Argument Gravity”

If set, all lines entered using the left button whose endpoints are near an actual terminal will be



snapped to that terminal. This is especially useful when editing schematic diagrams (`vem` automatically turns this option on when the edit style is set to schematic). The .Xdefault parameter *vem.gravity* specifies the maximum distance between line endpoints and terminals for gravity to have an effect (by default, 10 pixels).

“Show Instance Bounding Boxes”



Options editor for init.pal:schematic

□ Expert mode	Snap (oct units)	25
■ Visible Grid	Major Grid Spacing (oct units)	10
■ Dotted Grid	Minor Grid Spacing (oct units)	10
■ Manhattan Argument Entry	Log Grid Base	0
■ Argument Gravity	Log Grid Minimum Base	1
□ Show Instance Bounding Boxes	Minimum Grid Difference	16
■ Show Actual Terminals	Solid Fill Threshold	12
□ Expand Instances	Bounding Threshold	5
■ Visible Title Bar	Abstraction Threshold	0
Oct units per Lambda 1	Interface Facet	-

FIGURE 19-1: The “window options” dialog box in vem.

If this option is set, `vem` will display bounding boxes around all instances. The instance name will be displayed in the center of these bounding boxes.

"Show Actual Terminals"

When viewing very large designs, drawing the highlighting around actual terminals can be expensive. If this option is turned off, `vem` will not draw highlighting around actual terminals.

"Expand Instances"

If on, the contents facet of instance masters will be displayed. Otherwise, the interface facet will be displayed. This has the same effect as the *toggle-expansion* command.

"Visible Title Bar"

If this option is set, `vem` will display its own title bar above each graphics window. If the option is turned off, the title bar will be turned off.

"`Oct` units per Lambda"

This parameter specifies the number of `oct` units per lambda. The output of the where command and the coordinate displays in the title bar are displayed according to the value of this parameter. By default, `vem` uses 20 `oct` units per lambda.

"Snap"

All graphic input into the window will be snapped to multiples of this parameter (given in `oct` units). By default, `vem` snaps to one lambda (20 `oct` unit) intervals.

"Major Grid Spacing"

This parameter specifies the grid spacing of major grid lines in `oct` units. If logarithmic grids are turned on, it specifies a multiplying factor for major grid line spacing (see Log Grid Base below).

"Minor Grid Spacing"

This parameter specifies the grid spacing of minor grid lines in `oct` units. If logarithmic grids are turned on, it specifies a multiplying factor for minor grid line spacing (see Log Grid Base below).

"Log Grid Base"

If non-zero, this option selects a logarithmic grid. Normally, there are two grids drawn at a fixed number of `oct` units (major and minor grid lines). In a logarithmic grid, grids are drawn at some constant (specified by the Major Grid and Minor Grid parameters above) times the nearest integral power of the grid base. For example, if the constant is two and the base 10, grids will be drawn at 2, 20, 200, etc.

"Minimum Grid Threshold"

This parameter specifies the smallest allowable space (in pixels) between grid lines before `vem` stops drawing them.

“Log Grid Minimum Base”
: This parameter specifies the smallest base to be used for drawing logarithmic grids (see above).

“Solid Fill Threshold”
: This parameter specifies the size (in pixels) before a shape is drawn using solid fill rather than stipple fill. A large number specifies all geometry should be drawn solid regardless of its size.

“Bounding Threshold”
: Label text drawn in bounding boxes (e.g. instances or terminals) may or may not be drawn depending on the size of the bounding box. This parameter specifies how many times wider the text may be than the box before the label is not drawn.

“Abstraction Threshold”
: This parameter specifies the maximum size of a bounding box (in pixels) where it is acceptable to draw a filled box rather than an outline to speed the drawing process.

“Interface Facet”
: This string parameter specifies the name of the displayed interface facet. This facet will be used to draw instances in unexpanded mode.

```
layer-display<*>No Arguments
```

This command posts the dialog shown in figure 19-2, which can be used to selectively turn on or off the display of any layer. At the bottom of the dialog are six control buttons. The *Ok*, *Dismiss*, *Apply*, and *Help* buttons work in the same way as described for *window-options*. The *All* button automatically selects all of the layers displayed in the body of the dialog. The *Clear* button automatically unselects all of the layers displayed in the body of the dialog. Above the control buttons there is a list of all layers dis-

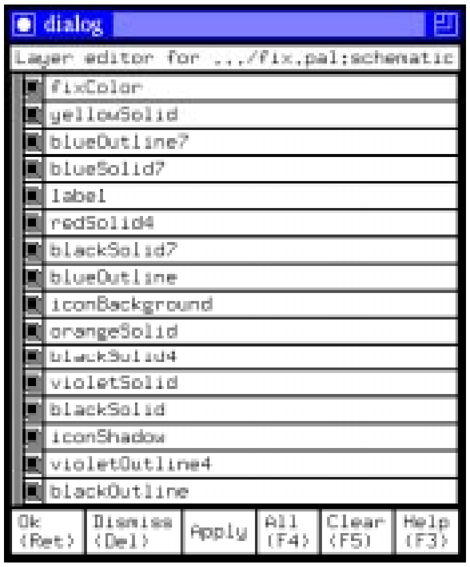


FIGURE 19-2: Layer display options in vem.

played in the window. The layers currently shown in the window have buttons to the left of the layer name that have check marks. Those that are not shown have buttons to the left of the layer name that appear empty. The state of a layer can be changed by moving the cursor over the corresponding button and depressing the left mouse button. Note that no window update will occur until either *Ok* or *Apply* are pressed.

19.5 Selection

The selection commands described below are used to manipulate object arguments on the argument list.

```
select-layer    .    [objs] [pnts] [lines] [boxes] "layer"
```

The *select-layer* command is similar to the *select-objects* command but allows the user to select only the geometries on a particular layer. This layer can be specified in two ways: the layer name can be typed in as the last argument or the command will try to determine the layer by looking at the geometry under the cursor when the command is invoked. If the spot for the layer is ambiguous, `vem` will post a dialog presenting a choice between the layers.

```
select-objects  s    [objs] [pnts] [lines] [boxes] "layer"
```

The *select-objects* command is used for placing collections of objects on the argument list for further processing by other commands. It takes as arguments any number of points, lines, or boxes. Points select items under the point, lines select objects which cross the line, and boxes select objects inside the box. If *select-objects* is not given any point, line, or box arguments, it will try to select items under the cursor where the command was invoked. These semantics are described in detail below.

For each point, the command adds zero or more of the objects under the point to the list. If there is more than one object under the point, a dialog will be posted with buttons representing each of the objects under the point. Clicking the mouse in one of the buttons highlights the object. Once the user has clicked on the desired objects, the *Ok* button is used to actually select the items.

For each line argument, the command adds all objects which intersect the line. This is useful for schematic drawings where paths (wires) are zero width. Selection using lines works best if the entered lines are Manhattan. Non-Manhattan lines may select more objects than intended.

For each box, the command adds all objects completely contained in the box to the list. Note that an object is considered contained if and only if its *bounding* box is *completely* inside the given box. The *select-objects* command is incremental; i.e. it may be called many times, each time adding to the selected set. All items selected are highlighted in the `vem` highlight color.

```
select-terms    ^T        [objs][points][lines][boxes]
```

This command selects all terminals (both actual and formal) whose implementations intersect the objects found by examining the argument list. The semantics for specifying the objects is identical to that described for *select-objects*. The items on the argument list will be replaced with the set of terminals found by examining the items. This command is useful for deleting formal terminals, specifying actual terminals for use by *edit-property* or *select-bag*, or creating new symbolic formal terminals using *promote.* As such, *it is extremely dangerous in Ptolemy.*

```
transform        t          [objs]
```

This command takes a selected set of objects built by selection commands and transforms them. The objects remain on the selected set. It is important to note that the actual objects in the database are not affected by this command. The transformation is associated with the object set not with the objects themselves.

The command takes (up to) three arguments: a set of objects to work on, a text rotation specification, and two points indicating a relative translation. The object set must be supplied. The rotation specification is a list of keywords, enclosed in quotation marks, separated by colons:

“mx”	Mirror around the Y axis.
“my”	Mirror around the X axis
“90”	Rotate counter-clockwise 90 degrees.
“180”	Rotate counter-clockwise 180 degrees.
“270”	Rotate counter-clockwise 270 degrees.

If both a rotation and a translation are given, the rotation specification should come first. If no rotation and translation are given, the command rotates the items 90 degrees counter-clockwise.

The *transform* command is incremental. This means it can be applied many times with each command having a relative effect on the current selected set. For example, invoking *transform* without arguments twice is the same as invoking it once and specifying the rotation string “180”.

Once the command completes, the highlighted form of the objects will reflect the specified transformation. This can be used as a reference for further rotations or translations.

Translations are specified by two points. A relative translation is applied to the current transformation based on the vector formed by the two points. It is also possible to interactively specify the translation of a selected set. This is done by moving the cursor to a reference point and pressing, dragging, and releasing the right mouse button. While the right button is depressed, the selected objects will track the cursor motion. This can be used to drag items around interactively.

After completing a transformation with transform, the objects are not actually changed until a command that manipulates objects is invoked (e.g. move-objects, copy-objects, etc.). See the *move-objects* and *copy-objects* commands below for more information.

```
unselect-objects      u     [objs] [pnts] [lines] [boxes]
```

The *unselect-objects* command is used to remove items from a previously created object argument (*select-objects* operation). Any number of points, lines, and boxes may be specified. The semantics for mapping these arguments to objects is the same as *select-objects*. For each point, zero or more items beneath the point that are part of the selected set are removed. For each line, all items in the selected set intersecting the line are removed from the set. For each box, all items completely contained in the box that are part of the selected set are removed from the set.

It is important to note that this command is intended to be used to unselect small sets of already selected objects. To unselect all items on the argument list, use control-W or control-U.

19.6 Property and Bag editing

The `oct` Data Manager supports two kinds of annotation: properties and bags. Properties are named objects which may have an arbitrary value and can be attached to any other `oct` object. Properties are used by `pigi` to store parameters. Bags are named objects which can contain any number of other `oct` objects, and are not used by `pigi`. Bags are used to represent common collections of objects (instances for example) that can be accessed efficiently. There are a number of `vem` commands used to create, edit, and delete properties and bags. However, since only one of these is useful in `pigi`, only one is documented here.

```
show-property   <*>        [objs]
```

The *show-property* command produces a list of all the properties associated with the object under the cursor when the command was invoked (or all objects in the selected set). If there are many objects under the cursor, the command will present a dialog which lists each object. The user can select one by clicking the mouse in the check box next to the object name. The object will be highlighted. When the right object is found, the user can click *Ok* to show the properties attached to the object. If the cursor is not over an object and nothing is in the selected set, the properties attached to the facet are shown. The properties are shown in the form: xid: name (type) = value. The *xid* is the external identifier for the property and can be used in evaluated labels. The command also echoes the type of the object each property is connected to.

19.7 Physical editing commands

The physical editing style in `vem` is used by `pigi` for editing icons. The commands described below are available in addition to the common commands described in the previous section.

```
alter-geometry  a          [box] [lines] or [pnts]
```

This command replaces the box, path, or polygon under the cursor with the new specification supplied on the argument list. This can be used to “stretch” geometry or change their composition. For example, to make a box slightly larger, enter the slightly larger box onto the argument list, move the mouse over the old box, and invoke *alter-geometry.*

```
change-layer    l     [objs] or [pnts][lines][boxes] "layer"
```

The *change-layer* command detaches geometry from its current layer and attaches it to a different layer. The geometry can be specified either as an object set constructed by the selection commands, or directly by drawing points on them, drawing lines through them, or drawing boxes around them. Normally, the target layer is determined by looking at the geometry under the cursor when the command was invoked. However, if the last argument to the command is text, it is interpreted as the name of the target layer.

```
copy-objects    x          [objs] {pnt pnt}
```

The *copy-objects* command copies a set of objects from one place to another. The command takes an object argument that should contain a list of objects to be moved (this is built with *select-objects* and *unselect-objects*). The command assumes the object set has been transformed using the *transform* command or interactively dragged to a new location with the right mouse button. The command makes a copy of the objects which are transformed according to this translation. For example, to copy objects from one location to another, the user first selects the objects using *select-objects,* then interactively drags the objects using the right button (transformation), then invokes the *copy-objects* command to make a copy at the new location. Since the items remain selected, new copies can be made without reselecting the objects.

The optional second argument should be two points which specify the source and destination points of the copy. This alternative can be used if the object set is too large for interactive dragging or one wishes to copy objects from one facet into another. If the copy is from one facet to another facet, terminals will not be copied and the objects will be copied in a manner that preserves the position of the objects relative to the source point. The default key binding for this command is short for “xerox”.

```
create-circle   C          [line] [pnts] "layer"
```

Since `vem` does not have a circle argument type, a special circle drawing command has been added. For most types of geometry, the user should use *create-geometry.* Circles are specified in one of two ways. The first is a line followed by up to two points. The line specifies a filled circle with the first point being the center and the second its outer radius. If the first point is supplied, an arc is assumed with an angle formed by the second point of the line, the center point and the newly specified point. The angle is measured counter-clockwise. If the last point is supplied, it specifies the inner radius of the circle (otherwise the inner radius is zero). The second form takes two points and an optional point. It specifies a circle where the inner and outer radius are the same. If the last point is supplied, an arc with the same semantics as the first form is assumed. Finally, the layer of the circle is determined from the cursor position or by a final text argument specifying the layer directly.

```
create-geometry c     [pnts] [lines] [boxes] [text] "layer"
```

The *create-geometry* command creates new geometry. It takes any number of points, lines, boxes, or text and a layer specification. A points argument creates a closed polygon. A line argument creates a multi-point path. Box arguments create boxes. Finally, text arguments create labels. When creating labels, the point set after the label is interpreted as the target points for the label. All the geometry is created on the same layer. This layer may be specified as the final text argument to the command or by invoking the command over an object attached to a layer. If the layer is ambiguous, the command will present a choice of layers in a dialog box. The palette command can be used to create a window which offers all possible layers for creating geometry.

```
create-instance       <*>        [pnt] {"master:view name"}
```

In most cases, the leaf cells designed with the physical editing style are not hierarchical. Instead, instances of the low-level cells are connected together using the symbolic editing style. However, those who would like to use `vem` as a purely physical design editor require instances in physical cells. This command places instances in the physical editing style.

The instance is placed relative to the point supplied to the command. The master of the instance can be specified in two ways. In the first form, the user supplies a text argument which contains the cell, view, and instance names separated by spaces. The instance name is optional. The second form determines the master of the instance by looking at the instance under the cursor when the command is invoked. This instance can be in the same cell or in another cell. A common practice is to build a cell of primitives and use this cell as a menu for placing physical instances (see the *palette* command).

```
create-terminal      <*>        ["term name"]
```

This command creates a new formal terminal named “term name”. The implementation of the terminal is determined by constructing a list of all geometries under the spot where the command was invoked and choosing the smallest coincident boxes from this list. *This command is dangerous in* `pigi`.

```
delete-objects  D         [objs] or [pnts] [boxes] "layer"
```

The delete command removes objects from a cell. The command has two forms. The first takes an object argument constructed using selection commands and deletes all of the items in this set. The second takes any number of point, line, and box arguments and a layer specification. This form deletes all objects under the points, all objects which intersect the lines, and all objects completely contained in the boxes that are attached to the specified layer. The layer may also be specified by placing the cursor over some other object attached to a layer when the command is invoked. If no layer is specified, all of the geometry is deleted.

```
edit-label      E         [pnt] {"LAYER"} or [objs]
```

The *edit-label* command creates and edits labels. The command has two forms. The first form creates a new label at the specified point on the given layer. If the layer is not specified, it will be determined by looking at the object under the cursor. The second form edits labels selected using the *select* command.

Labels in `oct` are represented as a box where the text is drawn entirely inside the box subject to justification and text height parameters. The edit-label command builds the box automatically by examining the text height and text itself. Thus, the user can control the justification, text height, and the label text. These parameters are set using the edit-label dialog box. This is a modeless dialog box that is posted when the user invokes the edit-label command.

The edit-label dialog box, shown on page 14, consists of three check-box areas for adjusting the label justification, and two type-in fields for adjusting the text height and the text itself. The justifications are computed in relation to the point the user specified when the label was first created. Thus, the horizontal justification specifies whether the point should be to the left, center, or right of the text. Similarly, the vertical justification specifies whether the point should be on the bottom, center, or top of the text. Finally, the line justification specifies how the lines should be justified within the text block when there is more than one line. The text height of the text is given in `oct` units. Note that the X window system does not directly support fully scalable fonts. Thus, `vem` uses a strategy where it will pick the closest font from a set of fonts that can be specified as a start-up parameter (see the document "Customizing Vem" for details). Finally, the last type in field can be used to enter the text for the label. The label can have as many lines as necessary.

At the bottom of the dialog are four control buttons: *Ok, Apply, Dismiss,* and *Help.* The *Ok* button updates the value of the label and causes the dialog to close. The *Apply* button updates the value of the label (showing the effects in the graphics window) but does not close the dialog. This allows the user to adjust the label several times if necessary. The *Dismiss* button closes the dialog without updating the value of the label. Finally, the *Help* button displays some help about how to use the dialog.

```
move-objects    m          [objs] {pnt pnt}
```

The *move-objects* command moves a set of objects from one place to another. The command takes an object argument that should contain a list of objects to be moved (this is built with *select-objects* and *unselect-objects*). The command assumes the object set has been transformed using the *transform* command or interactively dragged to a new location using the right mouse button. The command moves the objects to a new location based on this transformation. For example, to move objects from one location to another, the user first selects the objects using *select-objects,* then interactively drags the objects using the right button (transformation), then invokes the *move-objects* command to actually move the items to the new location. The items remain selected for further moves if necessary.

The optional second argument should be two points which specify the source and destination points of the move. This alternative can be used if the object set is too large for interactive dragging. Unlike the *copy-objects* command, objects cannot be moved from one facet to another.

19.8 Symbolic editing commands

The symbolic editing commands in `vem` allow the user to edit `oct` symbolic views. Symbolic views are used to represent layout in a form suitable for compaction and simulation. Since they are not used by `pigi`, they are not documented here.

19.9 Schematic editing commands

The schematic editing style is an extension of symbolic. In addition to the general, selection, and options editing commands, the following commands are specific to the schematic editing style:

```
delete-objects  D          [objs] or
                           [pnts, lines, boxes] ["layer"]
```

This command takes either an object list created with the *select-objects* and *unselect-objects* commands, or points, lines, and boxes with an optional layer-name. The resulting objects are deleted from the cell.

```
select-major-net      Control-N
                               pnts, lines, or boxes
                               ["net name"]
```

This command finds the net associated with the object under the points, intersecting the lines, or inside the boxes and highlights all objects on that net. If no points, boxes, or lines are specified, the object under the cursor will be examined. Alternatively, if a net name is provided, objects on the named net are highlighted. This command can be used to check the connectivity of a symbolic cell. The command is incremental (i.e. multiple nets can be selected).

```
move-objects    m          [objs] {pnt pnt}
```

The *move-objects* command in schematic editing mode is similar to the same command in physical editing mode above. One difference, however, is that the connectivity of the items moved by this command is not changed. This means an instance moved using this command also causes the segments attached to its terminals to move as well. Moving objects between facets is not supported.

```
copy-objects    x          (See Below)
```

The *copy-objects* command copies a set of objects from one place to another. This command has the same form as *move-objects* (see above) except that the objects are copied not moved. Like *move-objects,* the objects copied remain on the argument list for further move and copy operations. However, unlike *move-objects,* the connections to the objects in the selected set are not copied. Instead, the connectivity between the selected items is copied along with the objects.

```
create          c          (See Below)
```

The *create* command allows the user to add new objects to a schematic cell. Different arguments given to *create* will produce different objects.

Formal Terminals — "terminal-name [type] [direction]" : create

When *create* is given a single argument string, the name of a new formal terminal, a formal terminal is created. The implementation of the formal terminal is taken to be the actual terminal currently under the mouse (note: a connector terminal can also be used for this purpose). Since terminals in schematic may be quite small, this routine will try to find nearby terminals if it doesn't find a terminal directly beneath the cursor.

Formal terminal names must be unique within the cell. If a formal terminal of the given name already exists, `vem` will display a dialog box asking whether or not you wish to replace the old terminal.

Two optional pieces of annotation can be placed on the terminal: type and direction. The type can be one of SIGNAL, CLOCK, GROUND, SUPPLY, or TRISTATE. If not provided, SIGNAL will be assumed. The direction can be one of IN, OUT, or INOUT. If not specified, INOUT will be assumed. If either of these values are not provided in the terminal name specification, `vem` will post a dialog box containing fields for entering both the terminal type and direction. Pressing the left mouse button in the value area for these fields will cause a menu of the possible choices to appear. New values can be selected by releasing the mouse button over the desired value. Once appropriate values are selected, activating the *Ok* button at the bottom of the dialog will save the annotations. Activating the *Dismiss* option will leave the annotations unspecified. These annotations can be edited later using the edit-property command.

Instances — pnts [[master:view] [instance-name] : create

If the arguments to *create* are a number of points followed by an optional string, instances will be created with their origins at those points. If the name of the master is not specified textually, it will be inferred from the instance under the mouse.

The string argument has two parts: the instance specification and name. Both of these fields are optional. Specifying a null string is considered to be the same as no string at all. The instance specification is a master-view pair, such as “amp:schematic”. If this field is left out, the master is inferred from the instance under the mouse. Otherwise, an attempt is made to locate the instance by the master-view pair. If the name field is given, the instance will be given the specified name.

NOTES: Newly created instances whose actual terminals intersect actual terminals of other instances will be automatically connected. In this case, no path is required between the terminals. Rotated and mirrored forms of instances can be created by instantiating a new instance and using the transform and move-objects or transform and copy-objects commands.

Paths — lines : create

This command creates new segments for connecting together instance actual terminals. A new series of segments will be created on a predefined layer (WIRING). Connector instances will be placed automatically at all jog points. The width of the new path is always zero.

Normally, the schematic editing style has a feature turned on called “gravity”. When you draw segments with gravity turned on, `vem` will try to connect the segments to a nearby terminal if you miss a terminal by a small amount. This is useful when editing a large cell.

```
edit-label      E          [pnt] {"layer"} or [objs]
```

The *edit-label* command creates and edits labels, and is identical to the version in physical editing mode, documented above.

19.10 Remote application commands

The following commands apply only if a remote application, like `pigi`Rpc, is running.

```
kill-application     <*>   No Arguments
```

The *kill-application* command kills the remote application which has control of the window where the command was invoked. This can be used to terminate runaway remote applications. Note it has no standard menu or key bindings; it is a type-in command only. This command may not work on all machines.

```
rpc-any         r          "host pathname"
```

This command starts up a remote application which is not on the standard list of applications in the `vem` menu. “host” specifies the network location for the application and “path” specifies the path to the executable on “host”. If the host specification is omitted, the local machine is assumed.

```
rpc-reset        <*>        No Arguments
```

If an application terminates abnormally, `vem` may not recognize that the window no longer has an application running in it. This command forces `vem` to reset the state of the window so that new applications can be run in it.

19.11 Customizing Vem

The `oct` manual would be required only by programmers wishing to modify `pigiRpc`; it is available from the Industrial Liaison Program office, Dept. of EECS, UC Berkeley, Berkeley CA 94720 (`http://www.eecs.berkeley.edu/~ilp`).

The Postscript file `Vemcustom.ps` can be found in the `other.src` tar file in the Ptolemy distribution as `ptolemy/src/octtools/vem/doc/Vemcustom.ps`. This file describes some of the X resources that can be set in `vem`.

If you are trying to modify the look and feel of `vem`, see "X Resources" on page 2-54. For a fairly complete list of X resources, you can also look at the `defaults.c` and `defaults.h` files in the `ptolemy/src/octtools/vem/main` directory. These files can be found in the Ptolemy `other.src` tar file. If you are having font problems with vem, see "pigi fails to start and gives a message about not finding fonts" on page A-20.

19.12 Bugs

See also "Bugs in vem" on page A-33.

- Opening a facet that is inconsistent (either out of date or one with conflicting terminals) is not handled very gracefully.
- Bounding boxes may not be drawn if there is no geometry in the cell.
- The set path width command doesn't work if you use a palette to specify the layer.

Chapter 20. pxgraph — The Plotting Program

Authors: *David Harrison*

Other Contributors: *Joseph T. Buck*
Edward A. Lee

20.1 Introduction

The `pxgraph` program draws a graph on an X display given data read from either data files or from standard input if no files are specified. In Ptolemy, this program is invoked by several stars in several domains, and by the plot command in `pigi`. The program is also available for stand-alone use, independent of Ptolemy. `Pxgraph` can display up to 64 independent data sets using different colors and/or line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels, and a legend. There are options to control the appearance of most components of the graph.

`Pxgraph` is a slight variant of `xgraph`, modified to handle unusual IEEE floating-point numbers such as Inf and Nan, and to accept binary as well as ASCII input.

20.2 Invoking xgraph

The synopsis for stand-alone invocation of `pxgraph` is

`pxgraph` [options] [=WxH+X+Y] [-display host:display.screen] [file ...]

The options are explained below. When invoking `pxgraph` through a Ptolemy star, or the plot command, any of the options can be specified. Hence the full flexibility of the program is available to the user.

20.3 Detailed description

The input format is similar to the Unix command `graph(1G)` but differs slightly. The data consists of a number of *data sets*. Data sets are separated by a blank line. A new data set is also assumed at the start of each input file. A data set consists of an ordered list of points of the form `<directive> X Y`. The directive is either `draw` or `move` and can be omitted. If the directive is `draw`, a line will be drawn between the previous point and the current point (if a line graph is chosen). Specifying a `move` directive tells `pxgraph` not to draw a line between the points. If the directive is omitted, `draw` is assumed for all points in a data set except the first point where `move` is assumed. The `move` directive is used most often to allow discontinuous data in a data set. The name of a data set can be specified by enclosing the name in double quotes on a line by itself in the body of the data set. The trailing double quote is optional. Overall graphing options for the graph can be specified in data files by writing lines of the form `<option>: <value>`. The option names are the same as those used for specifying X resources (see below). The option and value must be separated by at least one space. An example input file with three data sets is shown below. Note that set three is not named, set two has discontinuous data, and the title of the graph is specified near the top of the file.

```
TitleText: Sample Data
0.5 7.8
1.0 6.2
"set one"
1.5 8.9
"set two"
-3.4 1.4e-3
-2.0 1.9e-2
move -1.0 2.0e-2
-0.65 2.2e-4

2.2 12.8
2.4 -3.3
2.6 -32.2
2.8 -10.3
```

After `pxgraph` has read the data, it will create a new window to graphically display the data. The interface used to specify the size and location of this window depends on the window manager currently in use. Refer to the reference manual of the window manager for details.

Once the window has been opened, all of the data sets will be displayed graphically (subject to the options explained below) with a legend in the upper right corner of the screen. To zoom in on a portion of the graph, depress a mouse button in the window and sweep out a region. `Pxgraph` will then open a new window looking at just that portion of the graph. `Pxgraph` also presents three control buttons in the upper left corner of each window: *Close*, *Hardcopy*, and *About*. Windows are closed by depressing a mouse button while the mouse cursor is inside the *Close* button. Typing EOF (control-D) in a window also closes that window. Depressing a mouse button while the mouse cursor is in the *Hardcopy* button causes a dialog to appear asking about hardcopy (printout) options. These options are described below:

“Output Device”	Specifies the type of the output device (e.g. “HPGL”, “Post-script”, etc.). An output device is chosen by depressing the mouse inside its name. The default values of other fields will change when you select a different output device.
“Disposition”	Specifies whether the output should go directly to a device or to a file. Again, the default values of other fields will change when you select a different disposition.
“File or Device Name”	If the disposition is “To Device”, this field specifies the device name. A device name is the same as the name given for the -P command of lpr(1). If the disposition is “To File”, this field specifies the name of the output file.
“Maximum Dimension”	

This specifies the maximum size of the plot on the hardcopy device in centimeters. `Pxgraph` takes in account the aspect ratio of the plot on the screen and will scale the plot so that the longer side of the plot is no more than the value of this parameter. If the device supports it, the plot may also be rotated on the page based on the value of the maximum dimension.

"Include in Document"
: If selected, this option causes `pxgraph` to produce hardcopy output that is suitable for inclusion in other larger documents. As an example, when this option is selected the Postscript output produced by `pxgraph` will have a bounding box suitable for use with psfig.

"Title Font Family"
: This field specifies the name of a font to use when drawing the graph title. Suitable defaults are initially chosen for any given hardcopy device. The value of this field is hardware specific -- refer to the device reference manual for details.

"Title Font Size"
: This field specifies the desired size of the title fonts in points (1/72 of an inch). If the device supports scalable fonts, the font will be scaled to this size.

"Axis Font Family and Axis Font Size"
: These fields are like *"Title Font Family*" and *"Title Font Size*" except they specify values for the font `pxgraph` uses to draw axis labels, and legend descriptions.

"Control Buttons"
: After specifying the parameters for the plot, the "Ok" button causes `pxgraph` to produce a hardcopy. Pressing the "Cancel" button will abort the hardcopy operation. Depressing the *About* button causes `pxgraph` to display a window containing the version of the program and an electronic mailing address for the author for comments and suggestions.

20.4 Options

`Pxgraph` accepts a large number of options most of which can be specified either on the command line, in the user's `~/.Xdefaults` or `~/.Xresources` file, or in the data files themselves. A list of these options is given below. The command line option is specified first with its X default or data file name (if any) in parenthesis afterward. The format of the option in the X defaults file is "program.option: value" where program is the program name (`pxgraph`) and the option name is the one specified below. Option specifications in the data file are similar to the X defaults file specification except the program name is omitted.

`=`*`W`*`x`*`H`*`+`*`X`*`+`*`Y`* `(Geometry)`
: Specifies the initial size and location of the `pxgraph` window.

`-`*`<digit> <name>`*
: These options specify the data set name for the corresponding data set. The digit should be in the range "0" to "63". This name

will be used in the legend.

`-bar` (`BarGraph`) Specifies that vertical bars should be drawn from the data points to a base point which can be specified with `-brb`. Usually, the `-nl` flag is used with this option. The point itself is located at the center of the bar.

`-bb` (`BoundBox`) Draw a bounding box around the data region. This is very useful if you prefer to see tick marks rather than grid lines (see `-tk`).

`-bd` *`<color>`* (`Border`)
This specifies the border color of the `pxgraph` window.

`-bg` *`<color>`* (`Background`)
Background color of the `pxgraph` window.

`-binary` This specifies that the input is a binary file rather than an ASCII file.

`-brb` *`<base>`* (`BarBase`)
This specifies the base for a bar graph. By default, the base is zero.

`-brw` *`<width>`* (`BarWidth`)
This specifies the width of bars in a bar graph. The amount is specified in the user's units. By default, a bar one pixel wide is drawn.

`-bw` *`<size>`* (`BorderSize`)
Border width (in pixels) of the `pxgraph` window.

`-db` (`Debug`) Causes `pxgraph` to run in synchronous mode and prints out the values of all known defaults.

`-fg` *`<color>`* (`Foreground`)
Foreground color. This color is used to draw all text and the normal grid lines in the window.

`-gw` (`GridSize`) Width, in pixels, of normal grid lines.

`-gs` (`GridStyle`) Line style pattern of normal grid lines.

`-lf` *`<fontname>`* (`LabelFont`)
Label font. All axis labels and grid labels are drawn using this font. A font name may be specified exactly (e.g. `9x15` or *`-*-courier-bold-r-normal-*-140-*`*) or in an abbreviated form: *`<family>-<size>`*. The family is the family name (like helvetica) and the size is the font size in points (like 12). The default for this parameter is `helvetica-12`.

`-lnx` (`LogX`) Specifies a logarithmic X axis. Grid labels represent powers of ten.

`-lny` (`LogY`) Specifies a logarithmic Y axis. Grid labels represent powers of ten.

`-lw` *`<width>`* (`LineWidth`)
: Specifies the width of the data lines in pixels. The default is two.

`-lx` *`<xl,xh>`* (`XLowLimit`, `XHighLimit`)
: This option limits the range of the X axis to the specified interval. This (along with `-ly`) can be used to "zoom in" on a particularly interesting portion of a larger graph.

`-ly` *`<yl,yh>`* (`YLowLimit`, `YHighLimit`)
: This option limits the range of the Y axis to the specified interval.

`-m` (`Markers`) Mark each data point with a distinctive marker. There are eight distinctive markers used by `pxgraph`. These markers are assigned uniquely to each different line style on black and white machines and varies with each color on color machines.

`-M` (`StyleMarkers`) Similar to -m but markers are assigned uniquely to each eight consecutive data sets (this corresponds to each different line style on color machines).

`-nl` (`NoLines`) Turn off drawing lines. When used with `-m`, `-M`, `-p`, or `-P` this can be used to produce scatter plots. When used with `-bar`, it can be used to produce standard bar graphs.

`-p` (`PixelMarkers`) Marks each data point with a small marker (pixel sized). This is usually used with the -nl option for scatter plots.

`-P` (`LargePixels`) Similar to -p but marks each pixel with a large dot.

`-rv` (`ReverseVideo`)
: Reverse video. On black and white displays, this will invert the foreground and background colors. The behavior on color displays is undefined.

`-t` *`<string>`* (`TitleText`)
: Title of the plot. This string is centered at the top of the graph.

`-tf` *`<fontname>`* (`TitleFont`)
: Title font. This is the name of the font to use for the graph title. A font name may be specified exactly (e.g. `9x15` or *`-*-courier-bold-r-normal-*-140-*`*) or in an abbreviated form: *`<family>-<size>`*. The family is the family name (like helvetica) and the size is the font size in points (like 12). The default for this parameter is `helvetica-12`.

`-tk` (`Ticks`) This option causes `pxgraph` to draw tick marks rather than full grid lines. The -bb option is also useful when viewing graphs with tick marks only.

`-x` *`<unitname>`* (`XUnitText`)
: This is the unit name for the X axis. Its default is “X”.

`-y` *`<unitname>`* (`YUnitText`)
: This is the unit name for the Y axis. Its default is “Y”.

`-zg` *`<color>`* (`ZeroColor`)
: This is the color used to draw the zero grid line.

`-zw` *`<width>`* (`ZeroWidth`)
: This is the width of the zero grid line in pixels.

Some options can only be specified in the X defaults file or in the data files. These options are described below:

`<digit>``.Color`	Specifies the color for a data set. Eight independent colors can be specified. Thus, the digit should be between ‘0’ and ‘7’. If there are more than eight data sets, the colors will repeat but with a new line style (see below).
`<digit>``.Style`	Specifies the line style for a data set. A string of ones and zeros specifies the pattern used for the line style. Eight independent line styles can be specified. Thus, the digit should be between ‘0’ and ‘7’. If there are more than eight data sets, these styles will be reused. On color workstations, one line style is used for each of eight colors. Thus, 64 unique data sets can be displayed.
`Device`	The default output form presented in the hardcopy dialog (i.e. `Postscript`, `HPGL`, etc.).
`Disposition`	The default setting of whether output goes directly to a device or to a file. This must be one of the strings `To File` or `To Device`.
`FileOrDev`	The default file name or device string in the hardcopy dialog.
`ZeroWidth`	Width, in pixels, of the zero grid line.
`ZeroStyle`	Line style pattern of the zero grid line.

20.5 Bugs

See “Bugs in pxgraph” on page A-35 for a list of `pxgraph` bugs.

Appendix A. Installation and Troubleshooting

Authors: *Joseph T. Buck*
Christopher Hylands
Alan Kamas

Other Contributors: *Stephen Edwards*
Edward A. Lee
Kennard White

A.1 Introduction

This appendix consists of three parts:

- "Obtaining Ptolemy" on page A-1 discusses how to obtain Ptolemy.
- "Ptolemy mailing lists and the Ptolemy newsgroup" on page A-2, discusses various forums for discussion about Ptolemy.
- "Installation" on page A-3 discusses how to install Ptolemy.
- "Troubleshooting" on page A-15 discusses how to find and fix problems in Ptolemy.
- "Known bugs" on page A-33 lists known problems in Ptolemy.
- "Additional resources" on page A-40 discusses other resources.

A.2 Obtaining Ptolemy

Ptolemy binaries are currently available for the following architectures: HP running HPUX10.20, Sun 4 (Sparc) running Solaris2.5.1, and Sun4 (Sparc) running SunOS4.1.3. In addition, Ptolemy has been compiled on HPUX9.x, Linux Slackware 3.0, NetBSD_i386, IBM RS/6000 AIX3.2.5, SGI Irix5.3, SGI Irix6.x and the DEC Alpha OSF1 V3.2 platforms. These additional platforms are not supported in-house and thus these ports are not tested and may not be complete.

Installing the full system requires about 150 Megabytes of disk space (more if you optionally remake). The demonstration version of Ptolemy, "Ptiny," only requires 12 Megabytes of disk space. All versions requires at least 8 Megabytes of physical memory.

For the latest information on Ptolemy, get the Frequently Asked Questions list. Send electronic mail to `ptolemy-hackers-request@ptolemy.eecs.berkeley.edu` with the message: `get ptolemy-hackers.FAQ` in the body (not the subject) of the message.

You may also send any questions on obtaining Ptolemy to the email address: `ptolemy@eecs.berkeley.edu`. If you have questions about using or enhancing Ptolemy, you should use the ptolemy-hackers mailing list. See "Ptolemy mailing lists and the Ptolemy newsgroup” on page A-2 for details.

A.2.1 Access via the Internet

Ptolemy is available *without support* via Internet FTP. Source code, executables, and documentation are all available on the FTP site at
`ftp://ptolemy.eecs.berkeley.edu`.

This site contains the latest release of Ptolemy, patches to the current release, a Postscript version of the Ptolemy manual, the demonstration version of Ptolemy, Ptolemy papers and journal articles, as well as the archives for the mailing list.

To obtain Ptolemy via FTP:

a. FTP to Internet host `ptolemy.eecs.berkeley.edu`.

b. Login as `anonymous`; give your full email address as the password.

c. Change to the `/pub` directory: `cd pub`

d. Follow the directions in the `README` file there.

There is an FTP mirror in Japan:
`ftp://ftp.iij.ad.jp/pub/misc/ptolemy`
The notation above means log into `ftp.iij.ad.jp` and then cd to */pub/misc/ptolemy*.

There is also an FTP mirror in France at:
`ftp://chinon.thomson-csf.fr/mirrors/ptolemy.eecs.berkeley.edu/`

A.2.2 Access via the World Wide Web

There is a World Wide Web (WWW) page for Ptolemy:
`http://ptolemy.eecs.berkeley.edu/`
The Ptolemy WWW page contains information about Ptolemy, demonstrations of Ptolemy programs, access to the Ptolemy FTP site, and a hypertext version of the Ptolemy Documentation.

There is a WWW mirror of our site in France at:
`http://chinon.thomson-csf.fr/ptolemy`

A.2.3 Obtaining documentation only

All of the manuals as well as a number of Ptolemy-related papers and journal articles are available from the WWW site and from the FTP site in the `/pub/ptolemy/papers` directory. These documents are all in the Postscript format and require a Postscript printer or viewer. The paper `overview.ps.Z` gives an overview of Ptolemy and would be of particular interest to new users.

A.3 Ptolemy mailing lists and the Ptolemy newsgroup

There are two publically available mailing lists and a newsgroup that discuss Ptolemy.

A.3.1 Ptolemy mailing lists

The Ptolemy mailing lists are run by the Majordomo mailing list server. This server can automatically subscribe you to mailing lists and it can send you monthly archive files for each of the lists. To find out more about our Majordomo server, send an email letter to: `majordomo@ptolemy.eecs.berkeley.edu` with the word `help` in the body of the letter.

There are two mailing lists for Ptolemy users:

`ptolemy-hackers@ptolemy.eecs.berkeley.edu`

`ptolemy-interest@ptolemy.eecs.berkeley.edu`

`ptolemy-hackers`

A forum for Ptolemy questions and issues. Users of the current release who have a Ptolemy question, comment, or think they've found a bug should send mail to `ptolemy-hackers`. Ptolemy users in companies and universities around the world who are interested in discussing Ptolemy post and read from the `ptolemy-hackers` group.

To subscribe to the `ptolemy-hackers` mailing list, send mail to:
`ptolemy-hackers-request@ptolemy.eecs.berkeley.edu`
with the word `subscribe` in the body of the message. To leave the mailing list put the word `unsubscribe` in the body of your message.

`ptolemy-interest`

A mailing list for Ptolemy announcements. Messages about new releases, bug fixes, and other information of interest to all Ptolemy users is posted here. Note: This is not a discussion group, and you cannot post to this mailing list. Message volume is very light. All users of Ptolemy should belong to this group in order to stay current with work being done on Ptolemy. Announcements sent to `ptolemy-interest` are also sent to `ptolemy-hackers`, so you need not subscribe to both lists.

To subscribe to the ptolemy-interest mailing list, send mail to:
`ptolemy-interest-request@ptolemy.eecs.berkeley.edu`
with the word `subscribe` in the body (not the subject) of the message.

A.3.2 Ptolemy Newsgroup

There is a Ptolemy newsgroup: `comp.soft-sys.ptolemy`. Just like the `ptolemy-hackers` mailing list, the `comp.soft-sys.ptolemy` newsgroup is a forum of the discussion of Ptolemy questions, bug reports, additions, and applications. Note that all mail sent to the `ptolemy-hackers` mailing list is automatically posted to the `comp.soft-sys.ptolemy` newsgroup as well. The name is chosen to correspond to similar newsgroups for the Khoros and Matlab systems, which are also under `comp.soft-sys`.oTroubleshooting

A.4 Installation

Ptolemy is a large software system that relies on a properly configured software environment. This section will take you step by step through the installation of Ptolemy, including all of the basic information required to get from an FTP archive or distribution tape to being able to run the system. Note that this information may have changed after the manual was printed. Thus, the most up-to-date instructions are included in `$PTOLEMY/doc/html/install`.

Ptolemy is distributed in several forms:

- The Ptolemy group provides Ptolemy and Gnu tools binaries for Solaris2.5.1, SunOS4.1.3 and HPUX10.20. We use these three platforms for in-house development and testing, so they are more stable than other platforms.
- We also make binaries for other architectures available. These binaries have been contributed by users and have not been tested in-house. You can find these binaries at `ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/contrib`. The list of contributed binaries varies, but usually includes Linux, AIX and DEC Alpha.
- If your machine is not one of the three primary platforms, and we do not have contributed binaries for it, then you will have to do a compile from scratch. For more information see “Rebuilding Ptolemy From Source” on page A-10.

A.4.1 Location of the Ptolemy installation

The Ptolemy system uses the environment variable `$PTOLEMY` to locate the Ptolemy distribution. If you are rebuilding Ptolemy from sources without using any prebuilt Ptolemy binaries, then you can set `$PTOLEMY` to any location. If you are using prebuilt binaries, then there are a few issues concerning exactly to what value `$PTOLEMY` is set. The issues occur because certain facilities, such as shared libraries, the Gnu compiler and Tcl/Tk save certain pathnames at build time. If at run time, certain files cannot be found that were present at build time, then there will be various failures.

The Ptolemy binaries that we distribute were built with `$PTOLEMY` equal to `/users/ptolemy`. If you can create a symbolic link from `/users/ptolemy` to your Ptolemy distribution, then you may find using the prebuilt binaries easier. If you cannot create a link and are using prebuilt binaries, then be aware of the following points:

- To use Ptolemy, you will need to set your `PTOLEMY` environment variable to the location of the distribution. For example, if your Ptolemy distribution was in `/usr/local/ptolemy`, then under C shell (`/bin/csh`), you would type `setenv PTOLEMY /usr/local/ptolemy`.
- If you are using prebuilt binaries that were built with `$PTOLEMY` equal to `/users/ptolemy`, but your Ptolemy distribution is not at `/users/ptolemy`, then you may need to set `LD_LIBRARY_PATH` or `SHLIB_PATH` so that the binaries can find the appropriate shared libraries at run time. The `pigi` script will attempt to properly set `LD_LIBRARY_PATH` if it is not already set, but the `pigi` script is easily fooled. See `$PTOLEMY/.cshrc` for sample C shell commands that set the appropriate variable. See also “pigi fails to start up, giving shared library messages” on page A-17, and “Shared Libraries” on page D-1.
- If you are compiling Ptolemy using the prebuilt Gnu binaries that we provide, and the Ptolemy distribution is not at `/users/ptolemy`, then you will need to set some additional environment variables, see “Gnu Installation” on page A-7 for more information.

- If you are using prebuilt binaries, and the Ptolemy distribution is not at `/users/ptolemy`, then the `tycho` command may fail for you. For a workaround, see “tycho fails to start up, giving TCL_LIBRARY messages” on page A-18.

A.4.2 Basic Ptolemy installation

First note the approximate disk space requirements for Ptolemy. The numbers below are for the Solaris version of Ptolemy, but other distributions are similar:

- Ptolemy from `pt-0.7.src.tar.gz` and `pt-0.7.sol2.5.tar.gz`: 135 Megabytes
- Gnu binaries for Solaris: 25 Megabytes
- Ptolemy from `pt-0.7.src.tar.gz` and `pt-0.7.sol2.5.tar.gz` if you remake (optional): approximately 168 Megabytes
- Documentation Postscript Files (optional): approximately 30 Megabytes
- Gnu Binary and Sources (optional): 66 Megabytes
- Additional Sources (needed to port Ptolemy) (optional): 24 Megabytes
- Complete rebuild from `pt-0.7.src.tar.gz`, `pt-0.7.other.src.tar.gz` and `pt-0.7.gnu.tar.gz`: under Solaris 2.5.1: 400 Megabytes

Important Note: Although we distribute binaries, you must also install the source tree in `pt-0.7.src.tar.gz`. *Installing this src directory is not optional*. The `src` directory contains the Ptolemy source code, but it also contains the icons and interpreted Tcl code used in the user interface. If you absolutely must discard the source code, you can remove all files under `src` with extensions `.cc`, `.h`, or `.c`. For more information, see “Freeing up Disk Space” on page A-14.

A.4.3 The ptolemy user

The preferred way to install Ptolemy involves creating a fictitious user with the userid `ptolemy`, together with a home directory for the `ptolemy` user. Using the binaries is easier if the `ptolemy` user’s home directory is `/users/ptolemy`. Once the `ptolemy` user account has been created, log in or `su` to user `ptolemy`. If you do not wish to create a user called `ptolemy`, see below for an alternative.

A.4.4 Installation without creating a ptolemy user

The preferred installation technique, as indicated above, is to create a user called `ptolemy`. The reason for this is that running Ptolemy requires an appropriate user configuration. At minimum, the user’s path must be set up properly. The `ptolemy` user is also configured to run an X window manager (`twm`) with suitable X resources that are known to work well. In troubleshooting an installation, having the `ptolemy` user properly configured can be very valuable.

Ptolemy can be installed without creating a ptolemy user. If you do this, *every* Ptolemy user must set a `PTOLEMY` environment variable to point to the root directory of Ptolemy. The installation is the same as below, except that `~ptolemy` is replaced with `$PTOLEMY`.

A.4.5 Obtaining Ptolemy

Ptolemy is available via the Ptolemy Web site at `http://ptolemy.eecs.berkeley.edu`. The Ptolemy releases can be found under the `Releases` link. The Ptolemy release can also be found on the Ptolemy ftp site at `ftp://ptolemy.eecs.berkeley.edu/` follow the instructions in the `README` file, and be sure to download in binary mode.

Untarring the distribution

Go to the directory where you either saved the downloaded `*.tar.gz` files. These files have been compressed with the Gnu `gzip` program, a compression program from the Free Software Foundation. In order to uncompress the files, you need the program `gzcat.`The `gzcat` program is available via anonymous FTP from `ftp://ptolemy.eecs.berkeley.edu/pub/gnu`. Now proceed as follows:

```
a. gzcat pt-0.7.doc.tar.gz | ( cd ~ptolemy/..; tar xf - )
```

This uncompresses the documentation, changes directory to the parent of the `ptolemy` user, and then creates all of the documentation files.

```
b. gzcat pt-0.7.src.tar.gz | ( cd ~ptolemy/..; tar xf - )
```

This uncompresses the `src` tar file and creates the source files. You must not skip this step. Ptolemy depends on these files being present. Note that you may get a few warning messages during this and the following step about the tar program not being able to create some directories because they already exist. This is expected (the same directory is mentioned in several of the tar files), so you need not worry.

c. If you are running the SunOS4.1.3 version:
   ```
   gzcat pt-0.7.sun4.tar.gz | ( cd ~ptolemy/..; tar xf - )
   ```
   (If you are running SunOS4.1.3 `/bin/tar`, and you see messages about
   ```
        tar: read error: unexpected EOF
   ```
   then see "EOF messages while using tar on Suns" on page A-23)
   If you are running the HPUX 10.20 version:
   ```
   gzcat pt-0.7.hppa.tar.gz | ( cd ~ptolemy/..; tar xf - )
   ```
   If you are running the Solaris2.5.1 version:
   ```
   gzcat pt-0.7.sol2.5.tar.gz | ( cd ~ptolemy/..; tar xf - )
   ```
   If you are running another platform for which a binary has been provided:
   ```
   gzcat pt-0.7.platform.tar.gz | \
           ( cd ~ptolemy/..; tar xf - )
   ```
   This uncompresses the binaries and creates the executable files. Note that is possible to install binaries for multiple platforms on the same file system because different directories are used for each set of binaries. Just execute whichever of the above commands apply.

d. (Optional) If you are planning on porting Ptolemy, you will need the other sources file:
   ```
   gzcat pt-0.7.other.src.tar.gz| \
                         (cd ~ptolemy/..; tar xf - )
   ```
   The `other.src` tar file includes sources to a subset of `Octtools` (including `vem`), `Tcl/Tk` and `xv`. If you wish to rebuild Ptolemy completely from source,

then you will need this tar file. Alternatively, you may be able to use the Octtools libraries from a binary tar file.
If you are planning on recompiling Ptolemy, but are really short on disk space, and you already have Tcl/Tk (comes with most Linux installations) then you can download `pt-0.7.oct.src.tar.gz` instead of `pt-0.7.other.src.tar.gz`. Note that this file has not been extensively tested. For more information, see "Ptolemy and Tcl/Tk" on page A-13

e. (Optional) You no longer need the `*.tar.gz` files that your got from the FTP site or the tape. Remember to delete these files to free up disk space.

f. (Optional) The X11 program `xv` is used by some of the image processing demos. If you already have a version of `xv`, or you don't plan on doing any image processing, you can also remove the `xv` binary in `$PTOLEMY/bin.$PTARCH/xv`.

A.4.6 Special considerations for use under OpenWindows

Ptolemy was developed using the X11R4 through X11R6 distributions from MIT. Although Ptolemy runs fine under OpenWindows 2, there are problems with running the Ptolemy graphical interface with OpenWindows version 3.0 under SunOS4.x. Some users have had no problems at all, but others have had intermittent problems such as "bad match" errors. We believe this may be a problem with the X-server supplied with the OpenWindows 3.0, but the error is elusive and we have not yet tracked it down. These problems seem not to occur in OpenWindows 3.3 and later (OW3.3 was distributed with Solaris2.3).

In order for all utilities included with this distribution to work under OpenWindows 2, you must install the shared libraries for the Athena widgets (the freely redistributable widget set from the MIT X11 distribution), which are provided with this distribution under the `$PTOLEMY/athena.sun4` directory. To install them, become root and copy all files in that directory into `/usr/openwin/lib` (or, if you have installed OpenWindows in a non-standard place, into `$OPENWINHOME/lib`). If you do not wish to do this, you could leave them in place and have every Ptolemy user change their `LD_LIBRARY_PATH` environment variable to search `~ptolemy/athena.sun4` before `/usr/openwin/lib`. Consult the Unix manual entry for the `ld` program to learn more about `LD_LIBRARY_PATH`.

After installation, the `$PTOLEMY` directory will contain several scripts for starting up X11R6 (Xrun), OpenWindows with *olwm* (Xrun.ow), or OpenWindows with *twm* (Xrun.ow.twm).

If only have OpenWindows, and not X11R5 or X11R6 and you plan on rebuilding from source, you may need the libraries in `athena.sun4`. `athena.sun4` is part of the `pt-0.7.sun4.tar.gz` tar file.

A.4.7 Gnu Installation

If you are planning on extending Ptolemy by writing your own stars and you are using prebuilt binaries, you will need to install the same compiler that was used to build the prebuilt binaries. In particular, Ptolemy supports dynamic linking of newly defined stars. *Dynamic linking will not work, however, if the new stars are compiled with a different version of the compiler than that used for the rest of the system*. Thus, you must either use the same compiler that we used for this distribution of Ptolemy or you must recompile the entire Ptolemy system. Note that this is not a complete set of Gnu software.

It is also possible to build Ptolemy with other, non-Gnu C++ compilers, such as Sun-Soft's C++ compiler. We have built this release of Ptolemy with the following compilers:

`sol2.5.cfront`	Sun CC, version 4.0 (native) for Solaris2.5.1
`hppa.cfront`	HP-UX CC version 4[1]

This release has also been built by others with SGI Delta-C++ for Irix5.3 (`irix5.cfront`).

We do not distribute these non-Gnu C++ binaries. If you choose to use a non-Gnu C++ compiler, you must completely rebuild Ptolemy. The libraries in the tar files were produced by the Gnu C++ compiler and are not interoperable with code from other compilers. When you completely rebuild Ptolemy, you must be sure to first remove all previously compiled object files. Note that to compile Ptolemy with a non-Gnu C++ compiler, you will still need to use Gnu make.

Ptolemy 0.7 was built with gcc-2.7.2.2 and libg++-2.7.2. Note that Ptolemy uses the gcc and libg++ shared libraries, if you are using your own version of gcc and libg++, it must have been configured with --enable-shared. For further information, see "Can I use my own version of gcc and libg++?" on page A-28.

The Gnu compiler is dependent upon where it was built. The executable that we supply assumes that the compiler is installed in a directory called `/users/ptolemy`. If you do not wish to rebuild the compiler, then you must either install Ptolemy in this directory, or create a symbolic link from this directory to the actual directory in which Ptolemy is installed. If you cannot install such a link, then you will still be able to run Ptolemy, *but you may not be able to dynamically link new stars or recompile Ptolemy.*

Even if your Ptolemy installation in not in `/users/ptolemy` and even if you cannot create a link from `/users/ptolemy` to the actual location of Ptolemy, you may still be able to use this compiler by setting Gnu environment variables before using the compiler. We provide a script in `$PTOLEMY/bin/g++-setup` that sets these variables. If you will always be using the prebuilt Gnu compiler shipped with Ptolemy with these variables, you may want to edit your copies of `$PTOLEMY/bin/pigiEnv.csh` and `$PTOLEMY/bin/ptcl` and add the following line:

```
    source $PTOLEMY/bin/g++-setup
```

You should not set these variables if you are using a version of the Gnu compiler that is different from the version that we are shipping. If you are using a different version of the Gnu compiler, then you will probably need to do a complete rebuild for dynamic linking to work. Please note that setting the environment variables does not work in all installations and that creating a link is better. If you are running under SunOS4.1.x and are using the prebuilt Gnu binaries, please see "Sun OS4 specific bugs" on page A-39. Here are the Gnu environment variables that worked for Solaris2.5.1:

```
setenv GCC_EXEC_PREFIX \
```

1. On the HP, type "what /usr/bin/CC" to see what version you are running.

```
       $PTOLEMY/gnu/$PTARCH/lib/gcc-lib/$PTARCH/2.7.2/
setenv C_INCLUDE_PATH $PTOLEMY/gnu/$PTARCH/lib/gcc-lib/$PTARCH
setenv CPLUS_INCLUDE_PATH \
$PTOLEMY/gnu/$PTARCH/lib/g++-include:\
        $PTOLEMY/gnu/$PTARCH/$PTARCH/include
setenv LIBRARY_PATH $PTOLEMY/gnu/$PTARCH/lib
```

The above assumes that the environment variable `PTOLEMY` is set to the name of the actual installation directory of Ptolemy, and `PTARCH` is set to the type of workstation (such as `sun4`, `hppa`, etc.).

To install the Ptolemy Gnu subset, proceed as follows:

a. Change to the directory that contains the files you downloaded via FTP or the Web.

You should now have a `pt-0.7.gnu.`*`xxx`*`.tar.gz` file (where *`xxx`* is an architecture supported by Ptolemy such as `sun4`, `hppa`, or `sol2`) in your current directory.

b. If you are running on a SunOS4.1.3 then:
```
gzcat pt-0.7.gnu.sun4.tar.gz| \
            (cd ~ptolemy/..; tar xf - )
```
(If you are running SunOS4.1.3 `/bin/tar`, and you see messages about
```
     tar: read error: unexpected EOF
```
then see "EOF messages while using tar on Suns" on page A-23)
If you are running on an HP workstation under HPUX10.20 then:
```
gzcat pt-0.7.gnu.hppa.tar.gz | \
            (cd ~ptolemy/..; tar xf - )
```
If you are running the Solaris2.5.1 version:
```
gzcat pt-0.7.gnu.sol2.5.tar.gz | \
            ( cd ~ptolemy/..; tar xf - )
```
If you are running another platform for which we provide a tar file:
```
gzcat pt-0.7.gnu.platform.tar.gz | \
             ( cd ~ptolemy/..; tar xf - )
```
Note that these are single commands split over two lines for readability. Do not type a space after the backslash at the end of the first line, just press `Return`.

c. There is also a Gnu tar file which contains the Gnu source code. You will save disk space and Ptolemy will still run if you do not untar the Gnu source code. However, if you plan to redistribute the Gnu tools (give them to anyone else) you must include sources, according to the Gnu Public License. Therefore, it may be a good idea to keep these source files around.
If you want to untar the Gnu source code:
```
gzcat pt-0.7.gnu.tar.gz | ( cd ~ptolemy/..; tar xf - )
```

d. You no longer need the `*gnu*.tar.gz` files that your got from the FTP site or the tape. You may delete these files to free up disk space.

A.4.8 Testing the Installation

Note that the following tests assume that you have created a `ptolemy` user and installed the system there. One advantage of such an installation, is that the `ptolemy` user already has a working `.cshrc` and `.login` file to make start-up easier.

To test Ptolemy, assuming you have set up a `ptolemy` user:

a. `login` as ptolemy

If the X server is not already running, the `.login` script will attempt to start it. If your installation is different from ours, you may need to modify `.login` to work at your site (in particular, you may need a different *path* variable).

b. `cd demo`

c. `pigi`

Follow instructions in “Starting Ptolemy” on page 2-2.

If you have not set up a `ptolemy` user, then set your `PTOLEMY` environment variable to point to the installation directory. If your Ptolemy distribution is at `/users/myptolemy`, under the C shell, you would type:

```
setenv PTOLEMY /users/myptolemy
```

If you use a shell other than `csh`, consult your documentation on how to set environment variables. The next steps are to change to the Ptolemy directory and to start up Ptolemy:

```
cd $PTOLEMY
bin/pigi
```

Note that the ptolemy user provides a model of a user properly configured to run Ptolemy. All the dot files (`.cshrc`, `.login` etc.) in the home directory are set up according to the tastes of the Ptolemy authors and according the standard use of windowing software in the Ptolemy development group.

A.4.9 Rebuilding Ptolemy From Source

If you wish to rebuild Ptolemy from source (this step is recommended if you plan to do major development work, such as adding a new domain), it is simply a matter of editing the appropriate configuration file and typing `make`. This is explained in a bit more detail below. Note that to rebuild completely from source, you need the `pt-0.7.other.srcs.tar.gz` tar overlay, as well as the `pt-0.7.src.tar.gz` tar overlay. If you are having problems rebuilding, you may want to look over “Ptolemy will not recompile” on page A-27.

To do a build of all of Ptolemy using the Gnu compiler from the distribution, first make sure that either Ptolemy is installed in `/users/ptolemy`, or that there is a symbolic link from `/users/ptolemy` to the installation directory. Alternatively, you can try setting the four environment variables described in “Gnu Installation” on page A-7.

`$PTOLEMY` should point to the location of the installation so that the toplevel makefile is at `$PTOLEMY/makefile`. `$PTARCH` should be set to the name of the architecture you are running. `$PTARCH` is used to select a makefile from `$PTOLEMY/mk/config-$PTARCH.mk`. The script `$PTOLEMY/bin/ptarch` will return the architecture of the machine on which it is run.

Next make sure that `$PTOLEMY/bin.$PTARCH` and `$PTOLEMY/bin` are both in your path (`$PTOLEMY/bin.$PTARCH` is where the compiler is installed. `$PTOLEMY/bin` is where certain scripts used to build star lists are located).

Then proceed as follows:

a. `setenv PTOLEMY /users/ptolemy`

b. `setenv PTARCH ‘$PTOLEMY/bin/ptarch‘`

c. `set path = ($PTOLEMY/bin $PTOLEMY/bin.$PTARCH $path)`

d. `cd $PTOLEMY` (or `cd /users/ptolemy`)

The toplevel makefile at `$PTOLEMY/makefile` can rebuild some or all of Ptolemy.

- To rebuild the Gnu tools, Tcl/Tk, `xv`, Octtools and Ptolemy, type `make bootstrap`. This target requires the following files: `pt-0.7.gnu.tar.gz`, `pt-0.7.other.src.tar.gz` and `pt-0.7.src.tar.gz`.
- To rebuild Tcl/Tk, `xv`, Octtools and Ptolemy, type `make everything`. This target requires the following files: `pt-0.7.other.src.tar.gz` and `pt-0.7.src.tar.gz`.
- To rebuild just Octtools and Ptolemy, type `make install_octtools install`.This target requires the following files: `pt-0.7.other.src.tar.gz` and `pt-0.7.src.tar.gz`. (If you are short on disk space, you may be able to download `pt-0.7.octtools.tar.gz` instead of `pt-0.7.other.src.tar.gz`).
- To rebuild just Ptolemy, type `make install`.This target requires the following files: `pt-0.7.src.tar.gz`.

An easier approach is to log in as “ptolemy” (assuming you created such a user). The path and environment variables are already set up for this user. Yet a third approach is to make use of the setup for this user, as follows:

a. `cd $PTOLEMY` (or `cd /users/ptolemy`)

b. edit `$PTOLEMY/.cshrc` to define `PTOLEMY` correctly

c. `source $PTOLEMY/.cshrc`

d. `make install`

If you wish to customize your installation, you may have to edit the configuration files. These files are in `$PTOLEMY/mk`. The configuration files are all named `config-$PTARCH.mk` where the `$PTARCH` is something like `sun4` for a Sun Sparc system running SunOS4.1.3 or `hppa` for an HP Precision Architecture machine. They are included by other makefiles and define symbols specifying compiler flags, the directory where X include files are located, etc.

If you wish to rebuild using a non-Gnu C++ compiler rather than `g++`, use `config-sol2.5.cfront.mk` as a starting point to produce your configuration file. This has been tested with Sun CC version 4.0 with Solaris2.5.1. For other platforms, you may need to do some tweaking. See `config-hppa.cfront.mk` for using HP CC version 4, and `config-irix5.cfront.mk` for using SGI Irix CC. Note that the term “cfront” is historical, and that not all of these compilers are actually cfront based. We use the term cfront to refer to non-Gnu C++ compilers.

To rebuild the system, first adjust the configuration parameters in the appropriate configuration file. For example, if you are using the Gnu tools on a Sun-4 running SunOS4.1.3, then you will need to adjust the `config-sun4.mk` file.

Next, run `make`. The Ptolemy source files include extensions found only in Gnu `make`, which is included in the Gnu subset of the Ptolemy distribution. (Make sure that the Gnu tools are installed correctly.) Sun `make` will fail on certain makefiles that have Gnu `make` extensions. See `$PTOLEMY/src/gnu/README` for a discussion of Gnu `make` compatibility.

You will get some warnings from the compiler, but the following warnings can safely be ignored:

- any warning about `file_id` defined but not used.
- any warning about `SccsId` defined but not used
- variable might be clobbered by `longjmp` or `vfor'`
  This warning may be a problem under heavy optimization. We will work to remove these from future releases.
- many warnings about `cast discards const`.

Details of how Ptolemy is built and the general layout

Below we describe some of the details of how Ptolemy is built from sources.

To build Ptolemy, you must have your `PTOLEMY` and `PTARCH` environment variables set. `PTOLEMY` is set to the location of the Ptolemy tree, `PTARCH` is set to the name of the machine architecture (the script `$PTOLEMY/bin/ptarch` will return the architecture of the machine on which it is run). The directory `$PTOLEMY/mk` contains master makefiles that are included by other makefiles (The makefile `include` directive does this for us). `$PTOLEMY/mk/config-$PTARCH.mk` refers to the makefile for the architecture `$PTARCH`. For instance, `$PTOLEMY/mk/config-sun4.mk` is the makefile that contains the sun4 specific details.

When you change to the `$PTOLEMY` directory and type `make`, `$PTOLEMY/makefile` contains a rule that checks to see if the directory `$PTOLEMY/obj.$PTARCH` exists. If this directory does not exist, then make runs the command `csh -f MAKEARCH`, where `MAKEARCH` is a C shell script at `$PTOLEMY/MAKEARCH`. `MAKEARCH` will create the necessary subdirectories under `$PTOLEMY/obj.$PTARCH` for `$PTARCH` if they do not exist.

We split up the sources and the object files into separate directories in part to make it easier to support multiple architectures from one source tree. The directory `$PTOLEMY/obj.$PTARCH` contains the platform-dependent object files for a particular architecture. The platform-dependent binaries are installed into `$PTOLEMY/bin.$PTARCH`, and the libraries go into `$PTOLEMY/lib.$PTARCH`. Octtools, Tcl/Tk, and Gnu tools have their own set of architecture-dependent directories. The Ptolemy Programmer's Manual describes the directory tree structure in more detail.

We are able to have separate object and source directories by using the `make` program's `VPATH` facility. Briefly, `VPATH` is a way of telling `make` to look in another directory for a file if that file is not present in the current directory. For more information, see the Gnu `make` documentation, in Gnu Info format files in `$PTOLEMY/gnu/common/info/make-*`.

Ptolemy and Tcl/Tk

Ptolemy0.7 uses `itcl2.2`, which is an object-oriented extension to tcl (Tool Command Language) and tk. `itcl2.2` includes a modified version of `tcl7.6` and `tk4.2`. If you have `itcl2.2` already installed, you may use your installed version. You will need to either edit
`$PTOLEMY/mk/config-default.mk` or create the proper links in `$PTOLEMY/tcltk`.

In theory, it is possible to build Ptolemy0.7 without `itcl2.2`. However, without `itcl2.2`, Tycho, the syntax manager and the Gantt chart facilities will not work. There may be other features that fail to operate. We strongly encourage you to build with `itcl2.2`.

In previous releases, the layout of the `$PTOLEMY/tcltk` directory was such that Tcl and Tk were in separate directories so that upgrading Tcl and Tk could be done separately if necessary. In Ptolemy 0.7, we are using `itcl2.2`, the Tcl and Tk are under the `$PTOLEMY/tcltk/itcl*` directories. We have left separate `$PTOLEMY/tcltk/tcl*` and `$PTOLEMY/tcltk/tk*` directories so that we can easily support separate Tcl and Tk releases in the future.

- `$PTOLEMY/tcltk/itcl` contains the architecture-independent Itcl directories `include`, `lib` and `man`. For instance, `tcl.h` might be at `~ptolemy/tcltk/itcl/include/tcl.h`.
- `$PTOLEMY/tcltk/itcl.$PTARCH` contains the architecture-dependent Itcl directories `bin` and `lib`. For instance, on the sun4, `libtcl.a` might be at `~ptolemy/tcltk/itcl.sun4/lib/libtcl.a`.

Notes for building on the sol2.5 platform (Sun4s running Solaris2.5.1)

Solaris2.5.1 is not shipped with a C compiler, and the Gnu tar file we ship does not include absolutely everything necessary to build on a compilerless Solaris2.5.1 machine. Notably, `bison` may be necessary to compile the Gnu C compiler and the Ptolemy program `ptlang`. If, when you are building the Gnu C compiler, `bison` is called and you do not have it, try using the Unix `touch` command on the offending .c file. The tar files we distribute include `ptlang.c`, which is generated from `ptlang.y`, so you should not need `bison` to compile Ptolemy. You can get a fairly complete set of Gnu tools via anonymous FTP from `ftp://ftp.uu.net/systems/gnu/solaris2.3/`.

- If you choose to compile Gnu `gcc` with SunSoft's `cc`, be sure that you are not using `/usr/ucb/cc`. Otherwise, you may see errors while compiling `gcc/protoize.c`:

  ```
  "/usr/include/sys/ucontext.h", line 25: syntax error before or
  at: stack_t
  ```

  The solution is to place `/opt/SUNWspro/bin` in your path before `/usr/ucb.`
- You will need`/usr/ccs/bin` in your path to pick up `ar`, `lex` and `yacc`.
- If you are building gcc-2.7.2 under Solaris, you must have `/bin` in your path before `/usr/ucb`. See `$PTOLEMY/src/gnu/README` for more information.
- Under `sol2.5.cfront`, `$OPENWINHOME` must be defined to build `xv` since `xmkmf` relies on `$OPENWINHOME`. In the C shell, one would type:

```
setenv OPENWINHOME /usr/openwin
```

A.4.10 Freeing up Disk Space

If you are short on disk space, you can consider removing the following files

- `$PTOLEMY/src/domains/sdf/demo/ppimage` (~1020 Kb). This file is used by some of the image processing demos.
- If you rebuild and reinstall Ptolemy from source, you may remove the `$PTOLEMY/obj.$PTARCH` directory once you are done installing.
- If you do not plan to rebuild Ptolemy from source, you can remove all the `.cc`, `.c` and `.h` files in the `$PTOLEMY/src` directory. Note that the `$PTOLEMY/src` directory contains the icons and interpreted Tcl code used in the user interface, so removing the `src` directory completely will prevent pigi from running.
- If you are very short on space, you may want to run Ptiny, a version of Ptolemy that only contains most of the SDF and DE domains. Ptiny is available via anonymous FTP from `ptolemy.eecs.berkeley.edu`. Note that Ptiny might be from an older release than the current release of the main Ptolemy distribution.

A.4.11 Other useful software packages

Some parts of Ptolemy use other software packages. The packages below are all available on the internet. If you have internet FTP access, you can find lots of FAQs (Frequently Asked Questions) via anonymous FTP at `pit-manager.mit.edu` in `/pub/usenet/news.answers`.

- The Utah Raster Toolkit (URT) is used by some of the Multimedia demos. Most of the URT utilities work with images in RLE (run-length encoded) format, so most of its utilities begin with the prefix `rle` such as is `rletoppm`. The original URT is available via anonymous FTP from `ftp.cs.utah.edu` in `/pub/urt-*.` Note that the original URT does not include configuration files for many modern platforms, including Solaris2.5.1. We have made URT sources and binaries for Solaris2.5.1 available via anonymous ftp from `ptolemy.eecs.berkeley.edu` in `pub/misc/urt`.
- The following Gnu utilities are available via anonymous FTP from `prep.ai.mit.edu`. See the file `/pub/gnu/GETTING.GNU.SOFTWARE` on that site. For further information, write to:

  Free Software Foundation
  675 Mass Ave
  Cambridge, MA 02139
  USA

  Ptolemy uses the following Gnu software:

`ghostscript -` PostScript previewer. Note that `oct2ps` can use `ghostscript` to generate Encapsulated PostScript (EPS).

`gdb` - needed for `pigi -debug`

`gzip` - Used to compress and uncompress files.

A.5 Troubleshooting

This section lists common difficulties encountered when installing and running Ptolemy. This list is, of course, by no means complete. If you do not find your particular problem here, refer to the section "Additional resources" on page A-40.

The most recent version of this section can be found on the bottom of the Ptolemy home page at `http://ptolemy.eecs.berkeley.edu/papers/almagest/appendixA.html`. The same file should be available via anonymous ftp from `ptolemy.eecs.berkeley.edu` as `pub/ptolemy/ptolemy0.7/TROUBLE_SHOOTING_0.7`.

A.5.1 Problems with tar files

EOF messages while using tar on Suns

There is a bug in the SunOS 4.1.3 version of `/bin/tar`. Sometimes a command such as:

```
gzcat pt-0.7.sun4.tar.gz | ( cd ~ptolemy/..; tar xf - )
```

may produce error message such as:

```
tar: read error: unexpected EOF
```

when reading from a pipe if the `tar` in the command is Sun's `/bin/tar`. One workaround is to use GNU tar, another is to use `gzcat` and `dd`:

```
gzcat pt-0.7.sun4.tar.gz | dd conv=sync,block |
                 ( cd ~ptolemy/..; tar xf - )
```

Another workaround is to uncompress the file first, and then run `/bin/tar`:

```
gzcat pt-0.7.sun4.tar.gz > pt-0.7.sun4.tar
cat pt-0.7.sun4.tar | (cd ~ptolemy/..; /bin/tar xf -)
```

A.5.2 Problems starting pigi

pigi: Command not found

Running the `pigi` command is the most common way of starting up Ptolemy. If you get a message like:

```
ptolemy@kahn 2% pigi
pigi: Command not found
ptolemy@kahn 3%
```

then try the following:

- The `pigi` script is located at `$PTOLEMY/bin/pigi`. If that file is not present, then you need to download the Ptolemy src file, `pt0.7.src.tar.gz`. This file is not optional, it contains the `pigi` script and other files necessary to run Ptolemy.
- Be sure that your path includes `$PTOLEMY/bin`. Under `csh`, do:

  ```
  set path = ($PTOLEMY/bin $PTOLEMY/bin.$PTARCH $path)
  ```

Mr. Ptolemy window does not come up

Ptolemy consists of two processes, `vem` and `pigiRpc`, that communicate via Remote Procedure Calls (RPC). `Vem` is the first process that starts up, and it produces a vem console window in the upper left corner of the screen and a green demo window just below it. When `pigiRpc` starts up, you should see a window in the middle of your screen that has the Mr. Ptolemy bitmap and a brief description of the binary you are running.

If the `pigiRpc` process fails to connect to the `vem` process, you won't see the Mr. Ptolemy bitmap, and the shift-middle-button menus will not be active. This problem seems to be most common on Linux machines, in part because they are often not on a network. If you are running under Linux and your installation is configured to use the network, then you may need to rebuild Ptolemy from source. See "Linux specific bugs" on page A-38 for more information.

If you are running on a machine that is not connected to a network, you will need to provide some network support for `pigi` to start up. `vem` and `pigiRpc` communicate with each other via RPCs, which require some intra-machine network support. One quick test is that you should be able to ping yourself:

```
/usr/etc/ping ‘hostname‘
```

There are several workarounds to this. One is to add the name of your host to the loopback line in `/etc/hosts` (here we add the name *myhostname*):

```
127.0.0.1  localhost myhostname
```

Under FreeBSD, you might have to add a fully qualified domain name. If you do not have a fully qualified domain name, sendmail might have problems. An example `/etc/hosts` entry would be:

```
127.0.0.1  localhost myhostname myhostname.mydomain
```

Another solution is to use `route` to route packets to your host through the loopback interface. As `root`, type:

```
route add ‘hostname‘ localhost 0
```

See the `ping`, `netstat` and `route` commands for more information about networking.

If the Mr. Ptolemy image fails to come up, another thing to check is that Tycho has the proper `tclIndex` files. If you try to open a facet by typing 'F', and you get a message like:

```
invalid command name "::tycho::Oct::openFacet"
    while executing
"::tycho::Oct::openFacet"
```

then check to see if the `TYCHO` environment variable is mis-set. You should be able to run Ptolemy with out setting `$TYCHO`, so if it is set, try `unsetenv TYCHO` and then restarting. Also, check to see that `$PTOLEMY/tycho/typt/kernel/tclIndex` exists. This file is used by `pigi` to find the `::tycho::Oct::openFacet` command at runtime. If it does not exist, create it by running `make sources` in that directory.

pigi fails to start when put in the background

A common problem occurs when `pigi` is started in the background and the user has the line

```
stty tostop
```

in their `.login` or `.cshrc` file. This command configures the terminal to halt any process that is running in the background when it tries to write to the terminal. One fix is to run `pigi` in the foreground. Another fix is to eliminate this command from your login files.

pigi fails to start up, giving shared library messages

On most platforms, Ptolemy is built using shared libraries. In Ptolemy 0.7, the Solaris2.x, HP and possibly Linux platforms use shared libraries, SunOS4.x does not. See the "Shared Library" appendix for more information about shared libraries.

At run time, if shared libraries cannot be found, you may see a message under HPUX-10.x like:

```
/usr/lib/dld.sl: Can't find path for shared library: libuprintf.sl
```

Under Solaris 2.x, you might see:

```
ld.so.1: /users/cxh/pt/bin.sol2/vem: fatal: librpcserver.so:
can't open file: errno=2
```

The message that you see may vary but the problem is that the binary cannot find the shared libraries to which it was linked. There are a few reasons this could be happening:

- The shared libraries are not where the binary expects them to be. If you are running from prebuilt binaries and your Ptolemy tree is not at `/users/ptolemy`, then this may be the problem.

  If you are running from prebuilt binaries and your Ptolemy distribution is not at `/users/ptolemy`, then you may need to set an environment variable to indicate what path should be searched for shared libraries. Under HPUX, the environment variable is `SHLIB_PATH`; under Solaris, the environment variable is `LD_LIBRARY_PATH`. The file `$PTOLEMY/.cshrc` should contain the proper commands to set the appropriate environment variable, though you may need to uncomment some lines. For HPUX, you could type the following command. (Do not type a space after the backslashes at the end of lines, just press `Return`):

```
setenv SHLIB_PATH {$PTOLEMY}/lib.{$PTARCH}:\
      {$PTOLEMY}/octtools/lib.{$PTARCH}:\
      {$PTOLEMY}/gnu/{$PTARCH}:\
      {$PTOLEMY}/tcltk/itcl.{$PTARCH}/lib/itcl
```

For Solaris, you could type the following command (all on one line):

```
setenv LD_LIBRARY_PATH {$PTOLEMY}/lib.{$PTARCH}:\
      {$PTOLEMY}/octtools/lib.{$PTARCH}:\
      {$PTOLEMY}/gnu/{$PTARCH}:\
      {$PTOLEMY}/tcltk/itcl.{$PTARCH}/lib/itcl
```

It is best to place these commands in your `~/.cshrc` file. It is possible that you might have to add other directories to the shared library path. For example, the Solaris2.x Ptolemy binaries are compiled with `/usr/openwin/lib` as the location of the X windows libraries. If your X windows libraries are in another directory, then you will need to add that directory to the shared library path. See "Window system problems" on page A-20.

- You do not have a library with that exact name. You may have an earlier or later version. Recompiling Ptolemy from scratch is one solution. It may also be possible to set up symbolic links to the proper libraries from a directory on the shared library path.

tycho fails to start up, giving `TCL_LIBRARY` messages

There are several ways to start up `tycho`, the Ptolemy syntax manager. `$PTOLEMY/bin/tycho` is a link to a script that processes command line arguments and starts up the appropriate binary. If you type `tycho -pigi`, `tycho` starts up with a binary that includes the Ptolemy system. If you type just `tycho`, then `tycho` starts up with the generic `itkwish` binary that is built from the `itcl` sources, which does not include any of the Ptolemy system.

If your Ptolemy distribution is not at `/users/ptolemy`, and you are running from prebuilt binaries, then if you run `tycho` with the prebuilt generic `itkwish` binary, you may see messages about:

```
application-specific initialization failed: can't find /users/
ptolemy/tcltk/itcl/lib/tcl7.4/init.tcl; perhaps you need to install
Tcl or set your TCL_LIBRARY environment variable?
```

What's happening here is the `itkwish` binary we ship has the `/users/ptolemy` path hard-coded into it and the binary is not finding the libraries it needs. The reason this happens is that we want `tycho` to be able to run outside of Ptolemy on machines that have only generic `itkwish` installed from the `itcl` distribution. There are a few workarounds:

- Create a link from `/users/ptolemy` to the Ptolemy distribution.
- `$PTOLEMY/bin/itkwish` is a link to a that will attempt to set your environment properly and then start the real `itkwish`. If you place `$PTOLEMY/bin` in your path before `$PTOLEMY/bin.$PTARCH`, then you will be running the `itkwish` script which might work for you.
- Run `tycho -ptiny` instead. The `tycho` script assumes that the Tcl installation is located in `$PTOLEMY/tcltk` if it is called with the `-ptiny`, `-ptrim` or `-pigi` arguments, so the script will do the right thing.
- Set environment variables in a script and then call `itkwish`:

```
#!/bin/sh
TCL_LIBRARY=$PTOLEMY/tcltk/itcl/lib/tcl
TK_LIBRARY=$PTOLEMY/tcltk/itcl/lib/tk
ITCL_LIBRARY=$PTOLEMY/tcltk/itcl/lib/itcl
ITK_LIBRARY=$PTOLEMY/tcltk/itcl/lib/itk
IWIDGETS_LIBRARY=$PTOLEMY/tcltk/itcl/lib/iwidgets2.0
export TCL_LIBRARY TK_LIBRARY ITCL_LIBRARY
export ITK_LIBRARY IWIDGETS_LIBRARY
exec $PTOLEMY/bin/tycho $*
```

For further information about troubleshooting Tycho, see the `$PTOLEMY/tycho/doc/troubleshooting.html`.

A.5.3 Common problems while running pigi

pxgraph fails to come up or displays a blank window

If the `pxgraph` program is given exceptional numbers, such as the IEEE floating-point `Inf`, `-Inf`, or `NaN` (“not a number”), then it will issue a cryptic error message “problems with input data” and will fail to display a plot. The stars that use `pxgraph` are supposed to intercept this and pop up an error message in a window. However, as of this writing, this does not work on all platforms. On such platforms, the error message, unfortunately, goes to the standard output, which may be buried several layers deep in your windowing system. It is also possible for the standard output to be lost, so that no error message appears. Thus, if you get such a `pxgraph` failure, look for instabilities in your Ptolemy schematic that would cause it to produce such exceptional numbers.

Old flowgraphs do not work (facets are inconsistent)

A `pigi` schematic contains references to icons. These icons are referenced by their location in the file system, typically using either an absolute path, a path relative to user’s home directory, or a path relative to the environment variable `PTOLEMY`. If the master for any of these icons is not in the expected place, `vem` will issue a warning, telling you the facet is inconsistent, and there will be blank space in place of the icon in the schematic. To find out what icon masters are missing, run the program `masters`. Instructions for doing this are given in “Copying and moving designs” on page 2-50. Invalid masters will be labeled “`INVALID`”. You must replace the invalid reference with a reference to a valid master. The tcl script `$PTOLEMY/bin/ptfixtree` can be useful for changing large numbers of facets. `$PTOLEMY/bin/ptfixtree.tcl` file contains limited instructions on how to use it.

One typical scenario for users upgrading from an earlier version of Ptolemy is that they will have references to `~ptolemy` in the directory tree. But the newer version may be installed somewhere else. One solution is to use the masters program to replace references to `~ptolemy` with `$PTOLEMY`.

Ptolemy simulations do not stop

In the SDF domain, it is possible to have multirate systems where a single iteration fires a very large number of stars. This happens when the number of samples produced or consumed by various connected stars in the system are mutually prime, or they have very large least common multiples. If a simulation is taking an unreasonable amount of time, then look for such mutually prime numbers (e.g, rates such as 53:97). Sometimes, in such circumstances, it can take a long time for the simulation to respond to pushing the "stop" button. It should, however, eventually respond.

Multi-porthole galaxies fail

If a galaxy contains a input or output multi-porthole, and the icon of the galaxy is named `Foo.input=2`, `Foo.output=2`, etc., the galaxy will fail to compile. This is because Ptolemy behaves as if anything ending in `input=`*X* or `output=`*X* must be a star. Avoid using names like `Foo.input=3` for galaxies.

Star is a compiled-in star and cannot be dynamically loaded

When you create a new universe (schematic), the domain assigned to the universe by default is SDF. If you create stars in the palette that are from another domain and then try to run the universe, you may get the error message

```
star 'Poisson' is a compiled-in star of domain DE. Cannot dynamically
load a compiled-in star class.
```

The solution is to change the domain of the universe by choosing edit-domain from the pigi menu (the keyboard short cut is 'd').

A.5.4 Window system problems

Below we discuss various problems we've seen between Ptolemy and the X window system.

Error: ld.so: libXext.so.4: not found

You have not installed the shared library needed by Ptolemy when it is used under OpenWindows. See "Special considerations for use under OpenWindows" on page A-7.

pigi fails to start and gives a message about not finding fonts

The default fonts for `vem` are specified in the file `$PTOLEMY/lib/pigiXRes9`, and also `pigiXRes9.bw` and `pigiXRes9.cp`. These files define a set of X window resources. The `pigiXRes9.bw` file is used if `pigi` is started with the `-bw` option. The `.pigiXRes9.cp` file is used in `pigi` is started with the `-cp` option. The definitions in these files can be overridden by the user. For example, a user who prefers to use microscopic fonts could set the X resource as follows:

```
Vem*font:  *-times-medium-r-normal--*-120-*
```

If, however, the fonts defined in these files on not available on the system, then the Ptolemy installer should change them in the files `$PTOLEMY/lib/pigiXRes9*`.

The fonts for Tk (and hence, the fonts for most of the dialog boxes) are specified in `$PTOLEMY/lib/tcl/ptkOptions.tcl`. These may similarly require modifications at some sites. In the worst case, if many standard fonts are not available, it may be necessary to redefine the default fonts built into the Tk source code, and recompile Tk. You may find the X11 program `xlsfonts` useful.

Ptolemy startup window only has an OK button

If the Ptolemy startup window does not have the Mr. Ptolemy bitmap and the copyright button, but instead the startup window is very small and has only an OK button, then you probably have font problems, see the section above for details about fonts.

Emacs confuses .pl files with Prolog

The `.pl` extension used to define Ptolemy stars is the same extension used for the Prolog language. Some text editors, such as Emacs, have special modes for editing Prolog files. These modes are inappropriate for editing Ptolemy files. You can add the following line to your `.emacs` file in your home directory:

```
(setq auto-mode-alist (cons '("\\.pl$" . c++-mode) auto-mode-alist))
```

Problems with the colormap

Some applications, for example FrameMaker 5, allocate as many colors as they can from the colormap when they start up. This may force applications that are started later (such as `pigi`) to have access to a very restricted set of colors. If when you start `pigi`, the welcome window appears in black and white, then you may have such a situation. If the situation is worse, and there are not enough colors in the colormap for `vem` to start, then you may not even get this far. One solution is simply to exit the offending application (e.g., FrameMaker or Netscape), and restart `pigi`. A better solution is to configure the offending application to use fewer slots in the colormap. We have found that for FrameMaker 5, the following X resources (placed in your `.Xdefaults` file) usually solve the problem:

```
        Maker.targetExactColors: 2
        Maker.minimumExactColors: 0
        Maker.targetColorCube: 4
        Maker.minimumColorCube: 1
```

This still leaves FrameMaker with a very rich set of colors to use. You may need to replace the 2 or 4 with smaller numbers if you have other color-intensive applications running (such as root window pictures of beaches in Tahiti).

The HP window system, VUE, may not display the correct colors when running Ptolemy. If the Vem window appears with white text on a tan background, or if the Ptolemy run window appears blue instead of tan, then VUE is getting the Ptolemy colors wrong.

The solution here is force VUE to use the regular Ptolemy X resources. Before starting `pigi`, in an xterm, do the following line:

```
xrdb -load $PTOLEMY/.Xresources
```

Then run `pigi`.

The window manager crashes

The window manager `twm` sometimes crashes when you are running Ptolemy. We do not know why. It seems to be an interaction with Tk. Our solution is to simply restart it. You may wish to make sure your configuration does not log you out when the window manager exits.

Problems with Sun Sparc5s with 24 bit TCX framebuffers

Some Sun Sparc5 machines have a 24 bit framebuffer called a TCX framebuffer. On these machines, `vem` will fail to start with a message like:

```
A Serious X Error has occurred:
         BadValue (integer parameter out of range for operation)
         Request X_QueryColors (minor code 0)
Type ‘y’ to continue, ‘n’ to exit, ‘a’ to abort:
```

The problem here is that the root window is a `TrueColor` window, which causes problems with `vem`. As a workaround, Arnaud LaPrevote suggests starting OpenWindows with

```
openwin -server Xsun -dev /dev/fb defclass PseudoColor
```

It seems you can safely ignore the warning message about specified class or parameter not available that occurs during startup. This bug does not occur on the 24 bit UltraSparc framebuffer, so it seems that the `vem` bug is only tickled by the tcx frame buffer. For more information, see `http://ptolemy.eecs.berkeley.edu/ptolemy0.7/html/sparc5tcx.html`.

Problems with Mac X and Ptolemy

Some people have had difficulties running Ptolemy with Mac X. Tze-Wo Leung of Bell Northern Research suggests the following setup for Mac X when using Ptolemy:

- My hardware setup is: Mac IIci w/ 20 MBytes DRAM, 210 MB HD and 21” two-page display/21gs Radius B&W monitor. The UNIX server is a HP 715 workstation.
- The ‘Display:’ option in the ‘Edit Remote Command’ MUST be set to ‘(2) Color Rootless’ mode.
- Increasing the memory allocation of Mac X to 4 MBytes also helps. The memory size can be increased by first exiting Mac X, then using the ‘Get Info’ option under the File menu to pop-up the file info window where one can change the memory allocation.
- In the control panel, the characteristics of the monitor are set to ‘Colors’ and ‘256’.

Problems with Exceed and Ptolemy

Under Hummingbird Exceed5.0, you may need to start up another X client before starting `pigi`. If you get an error message like:

```
Error: UGetFullTechDir: cannot read .Xdefaults ‘vem.technology’:
UserMain: OpenPaletteInit() failed
```

then try starting up another X client such as `xclock` before starting up `pigi`.

Problems with XFree86

XFree86 3.1 has problems with the `vem` Edit-Label widget. If you see messages like

```
A Fatal Xt Toolkit Error has occurred:
Attempt to unmanage a child when parent is not Composite
Type ‘y’ to continue, ‘n’ to exit, ‘a’ to abort
(continuing may have unpredictable consequences):
```

then the solution is to upgrade to XFree86 3.2.

A.5.5 Problems with the compiler

The first thing to try is compiling a 'hello world' program in C or C++. In C++, you should probably try using the stream functions, below is a sample file:

```
#include <stream.h>
main() { cout << "Hello, Ptolemy.\n"; }
```

Try compiling the file with `g++ -v` and `-H` flags turned on. `-v` tells you what steps the compiler is running, `-H` tells you what include files are being read in.

```
g++ -v -H hello.cc
```

Look at each step of the compilation, and pay particular attention to the assembler and loader steps. You can use the `-save-temps` gcc option to save any temporary files created in each step. Then, if necessary, you can try running each step by hand.

as vs. gas

`gcc` can use the native assembler or the GNU assembler. Often the GNU assembler is installed as '`as`'. Check your path to see which version you are getting. `gcc` can often be configured at compiler build time to use the native assembler or the GNU assembler (`gas`), but once the compiler is built, you are stuck with one or the other assemblers. The Ptolemy project makes GNU binaries available. Most of the GNU binaries that we distribute use the native assembler, that is, they don't use `gas`. However, the hppa GNU `gcc-2.7.2` binaries use `gas`. We distribute a `gas` binary with the hppa GNU `gcc-2.7.2` binaries.

Collect

To pick up C++ constructors and destructors, `g++` can use the native loader or a program called '`collect`' (for more information, see Joe Buck's g++ Frequently Asked Questions [FAQ] described below). We usually use `collect`, because it works with the Pure Inc. tools. The collector is usually located at `gcc-lib/$PTARCH/COMPILER_VERSION/ld`, e.g. the gcc-2.7.2.2 sun4 collector might be at `$PTOLEMY/gnu/sun4/lib/gcc-lib/sun4/2.7.2.2/ld`. Note that `g++` under Solaris2.x does not use `collect`.

You can pass the collector arguments so that it will print out more information. Try

```
g++ -v -Wl,-debug hello.cc
```

or, if you are within Ptolemy:

```
make LINKER="g++ -v -Wl,-debug"
```

If you pass `collect` the `-debug` flag, you will get a lot of output. Part of the output will include what binaries and paths collect is using. Below is part of the output the `collect -debug` generated by a working installation.

```
ld_file_name        = /usr/bin/ld
c_file_name         = /users/ptolemy/bin.sun4/gcc
nm_file_name        = /usr/sww/bin/gnm
strip_file_name     = /usr/tools/gnu/bin/gstrip
c_file              = /usr/tmp/cca01064.c
o_file              = /usr/tmp/cca01064.o
COLLECT_NAMES       = /users/ptolemy/gnu/sun4/lib/gcc-lib/
sun4/2.7.2.2/ld
COLLECT_GCC_OPTIONS = -v -L../../lib.sun4 -static -L../../
```

```
        octtools/lib.sun4 -L../../tcltk/tk.sun4/lib -L../../tcltk/
        tcl.sun4/lib -L/usr/X11/lib -o pigiRpc
        COLLECT_GCC          = gcc
```

If you need to change the `*_file_name` values, try modifying your `$path` so that the new program is in front of the program listed. For instance, if, under `csh`, one wanted to use `/usr/local/bin/nm` instead of `/usr/sww/bin/gnm` in `nm_file_name` above, one would type:

```
set path=($PTOLEMY/bin.$PTARCH $PTOLEMY/bin /usr/local/bin
$path)
```

The collector will also respond to certain environment variables, see the source in the `gnu` tar overlay at `$PTOLEMY/src/gnu/src/gcc/collect2.c`.

The collector creates a temporary file that has the constructors and destructors in it. To get collect to save the temporary file, set the following environment variable:

```
setenv COLLECT_GCC_OPTIONS -save-temps
```

If the collector is getting an old version of GNU `nm`, then you could have problems. Passing the collector the `-debug` flag might help here.

Error: Linker: no constructors in linked-in code!

If you see the above message while linking a new star, then you might be having `nm` problems. The above message seems to occur under Linux. Joe Buck says that the thing to do is:

The incremental linker is searching the object file for a global symbol that has the form of a constructor for a static or global object. It then calls that constructor. Ptolemy stars use these constructors to “register” themselves on the list of known stars. You can then create instances of your new star by using its `clone()` method. If it can’t find the symbols, then the new star’s code isn’t accessible.

The “`nm`” program is used to find the constructor symbols. Perhaps you have an older version of `nm` on your system? Find the `.o` file corresponding to your star and execute

```
/usr/bin/nm -g --no-cplus mystar.o | grep GLOBAL
```

(`--no-cplus` tells nm not to demangle the symbols). You should get some constructor and destructor symbols.

Environment variables

Certain environment variables can control where the compiler looks for subprograms and include files. These four variables are usually in the Ptolemy distribution so that users can run the prebuilt compiler, even if the distribution is not at `/users/ptolemy`.

```
setenv GCC_EXEC_PREFIX $PTOLEMY/gnu/$PTARCH/lib/\
gcc-lib/$PTARCH/2.7.2.2/
setenv C_INCLUDE_PATH $PTOLEMY/gnu/$PTARCH/lib/gcc-lib/$PTARCH
setenv CPLUS_INCLUDE_PATH \
$PTOLEMY/gnu/$PTARCH/lib/g++-include:\
$PTOLEMY/gnu/$PTARCH/$PTARCH/include
setenv LIBRARY_PATH $PTOLEMY/gnu/$PTARCH/lib
```

See the `gcc` info format file for a complete list of environment variables. Note that `GCC_EXEC_PREFIX` must have a trailing slash, "`/`". The above variables work for Solaris2.x, if you are running SunOS4.1.3, see "Sun OS4 specific bugs" on page A-39. One symptom of having improperly set environment variables is if you see messages about:

```
ptolemy/src/kernel/isa.h:44: conflicts with new declaration
with C linkage
```

Another symptom is if you have missing `strcmp()` symbols at link time.

If, under gcc-2.7.2.2, you get warnings about 'conflict with built in declaration', and your compiler is not installed where it was built, you may need to create a link in your gcc-lib. We have also seen problems with functions that have variable numbers of arguments. If you compile the file with the `-v` option, you can see what directories `gcc` is including. You could try creating a link:

```
(cd $PTOLEMY/gnu/$PTARCH/lib/gcc-lib/$PTARCH/2.7.2.2; \
     ln -s .. $PTARCH)
```

Using trace

The SunOS4.1 `trace` command can be invaluable in determining what a program is doing at run time. If you compile with `gcc -v -save-temps` then you can try running `trace` on the various steps, and see each system call. Unfortunately, the filenames are truncated, but often this is enough to see what is going on. Solaris has a similar `truss` command.

A.5.6 Problems compiling files

There are several ways to handle problems while compiling files. These problems are often caused by strange interactions between `.h` files, and they occur while compiling a particular file or a set of files. Note that these problems are different than problems that occur during link time, which we discuss in "Missing symbols while linking pigiRpc" on page A-29.

Using cpp to diagnose .h file problems

If you are having problems with include files, try modifying a hello world program (see above) to include those files. Note that you could be getting unexpected substitutions from the C preprocessor `cpp`, so looking at the `cpp` output can be useful in solving compiler installations problems and include file problems

The gcc `-E` and `-P` options are very useful in wading through include file problems. `-E` stops compilation after the C preprocessor runs, and outputs the resulting file. `-P` strips off the line numbers from the output.

Try using the `-E` option, and look at the output file. Sometimes the problem will be obvious. Note that if your compile arguments include `-o` *`filename.o`*, then *`filename.o`* will have `cpp` text output, not the usual object file. Note further that in some compilers, the `-c` option (create a `.o` file) will override the `-E` option. If `-c` does override `-E`, you will have to grab the output of the make command and place it in a temporary file, say `/tmp/doit`, edit `/tmp/doit` and remove the `-c` option and then type `sh /tmp/doit`. If `-c` does not override `-E`, and you are within Ptolemy, you can try using the `OPTIMIZER` makefile flag to pass arguments to the compile. For instance:

```
make OPTIMIZER=-E Linker.o > Linker.e
```

Another approach is to run `cpp` and then re-run the compiler on the `cpp` output. In `gcc,` the `-P` option strips out the `cpp #line` comments. You can use `-E -P` to generate a new file that has all the cpp substitutions in it, and then try compiling the new file:

```
make OPTIMIZER=”-E -P” Linker.o > tst.cc
```

Edit `tst.cc` and remove the first line, which will have the gcc command in it. Make `tst.o`:

```
make OPTIMIZER=”-v -H” tst.o
```

Using the `gcc` arguments `-E -dM` will tell you what symbols are defined by `cpp` at the end of the compile. See the `gcc` man page or the `gcc` info format file for more information.

Narrowing the problem down.

If you are having strange problems compiling one file, you might want to try to find the smallest file that causes the problem. This method can take time, but it is sometimes the only way to find a solution. One way is to wrap code in `#ifdef NEVER` ... `#endif` and narrow the bug down to one function. Changing the `#include` declarations at the top, and following the include file change can also help here.

Using c++filt to demangle symbols

When a C++ file is compiled, the symbol names found inside a `.o` file or a library file have been specially processed by the compiler. This special processing is called mangling. The symbol names may look unusual, for example, `makeNew__10KnownBlockPCcT1` is the mangled version of `KnownBlock::makeNew(char const *, char const *)`. You may find it useful to be able to convert the mangled symbol names back to the human readable C++ symbol name. Under gcc-2.7.2.2, you can use the `c++filt` program to do the conversion:

```
cxh@brahe 9% echo “makeNew__10KnownBlockPCcT1” | c++filt
KnownBlock::makeNew(char const *, char const *)
cxh@brahe 10%
```

On platforms where we distribute the GNU compiler, `c++filt` can be found at `$PTOLEMY/bin.$PTARCH/c++filt`.

Sources of information for compiler problems

`$PTOLEMY/gnu/common/man/man1/gcc.1` contains the gcc man page. This file is shipped with the prebuilt GNU binaries. You can try placing `$PTOLEMY/gnu/common/man` in your MANPATH environment variable:

```
setenv MANPATH $PTOLEMY/gnu/common/man:$MANPATH
```

`$PTOLEMY/gnu/common/info/gcc*` contains the GNU Info format documentation. Use Emacs (M-x info) or a program such as `tkinfo` to view the info pages (`tkinfo` is available via anonymous FTP from `ptolemy.eecs.berkeley.edu` in `pub/misc`).

`$PTOLEMY/src/gnu/g++FAQ.txt` is Joe Buck’s g++ Frequently Asked Questions document in text format, (g++FAQ and other FAQs are available via anonymous FTP from `rtfm.mit.edu` in `pub/usenet/news.answers`).

The following FAQs might also help: c.faq hpux.faq solaris2.faq solaris2_porting.faq sun_sysadmin.faq.

A.5.7 Generated code in CGC fails to compile

The `Makefile_C` target uses the Ptolemy makefile structure to determine platform dependencies in the C language Code Generation (CGC) domain compile command. If you are having problems with platform dependencies, you may want to use the `Makefile_C` target. Some demos, such as the CGC `commandLine` demo use the `Default-CGC` target. The `Default-CGC` target has the compiler name set as a target parameter, which is usually either `gcc` or `cc`. Unfortunately, not all machines are shipped with a working `cc` binary and not all machines have `gcc`, so we cannot choose a default that will work in all circumstances. If you do not have the compiler that is listed in the target parameter, you can do any of the following:

- Type a ‘T’ while in the facet to bring up the Target Parameters window and change the value of `compilerCommand` to a compiler that you have.
- Create a link in `$PTOLEMY/bin.$PTARCH` for the compiler you don’t have. For example, if you don’t have `gcc`, and your `cc` is at `/usr/ccs/bin/cc`, you could do

  ```
  cd $PTOLEMY/bin.$PTARCH; ln -s /usr/ccs/bin/cc gcc
  ```
- Use the `Makefile_C` target instead of the `Default-CGC` target. (Some demos still use the `Default-CGC` target so that we can continue to test that target).

The targets in CGC are configured by default with reasonable guesses about the compile and link options that are required to compile the code. However, the actual options required depend on your system configuration. For instance, your default `cc` compiler may not have been configured to automatically find the X11 include files. You might, therefore, get an error message the `Xlib.h` cannot be found. You should find out where on your system `Xlib.h` is installed, use the ‘T edit-target’ command to add a compile option of the form `-L`*path_name* where *path_name* is the full path of the directory containing the file. ‘T edit-Target’ is on the shift-middle-button menu.

Certain compilers will change their behavior depending on the values of certain environment variables:

- The hppa `cc` compiler will use the `CCOPTS` environment variable. If your X11 libraries were at `/usr/sww/X11R5/lib`, then one could exit `pigi`, set the variable `setenv CCOPTS -L/usr/sww/X11R5/lib` and restart `pigi`. Then when you compile CGC demos with hppa `cc`, you should be able to find the proper libraries. See the hppa `cc` man page for more information. Note that under HPUX10.x, the bundled C compiler is not ANSI compliant so it may fail to compile some CGC Universes.
- GNU `gcc` will also use certain environment variables. The `LIBRARY_PATH` variable may help:
  ```
  setenv LIBRARY_PATH /usr/sww/X11R5/lib
  ```
  See the `gcc` documentation for more information.

A.5.8 Ptolemy will not recompile

If Ptolemy fails to recompile, you may be using a substantially different version of the GNU compiler. The system is most likely to build if you use the same tools that we used originally. The GNU tools we used are supplied with the Ptolemy distribution. We discuss common Ptolemy compilation problems below. For further information about recompiling Ptolemy, see “Rebuilding Ptolemy From Source” on page A-10 and see Volume 3 of the Ptolemy Almagest, “The Ptolemy Programmer’s Manual”.

Messages about “unexpected end of line seen” while running make

If you are running a version of make other than GNU make, you may see messages like:

```
make: Fatal error in reader: ../../mk/stars.mk, line 52: Unex-
pected end of line seen
```

Ptolemy contains GNU make extensions, you must run GNU make to build Ptolemy, even if you are not using the GNU compiler. GNU make binaries are available via anonymous FTP in `ptolemy.eecs.berkeley.edu`. The Ptolemy binary tar files for hppa, sol2 and sun4 contain GNU make binaries. You can get just the GNU make binary in `pub/gnu/$PTARCH/make.gz`, where `PTARCH` is one of hppa, sol2 or sun4, or you can get the GNU make binary, along with the GNU compiler and other binaries in `pub/gnu/ptolemy0.7`.

Apparently, older versions of GNU make, such as 3.71, can fail with a message like:

```
../../mk/stars.mk:113: *** commands commence before first target.
Stop.
```

If you get such a message, type `make -v` to see what version of GNU make you are running.

Can I use my own version of Tcl/Tk?

Ptolemy 0.7 uses `itcl2.2`, which is an extension to Tcl/Tk. If you have `itcl2.2` already installed, you may use your installed version. See “Ptolemy and Tcl/Tk” on page A-13. Tycho will not work with `itcl2.1`, you must use `itcl2.2`. Tycho is necessary for viewing the contents of stars and other important features.

Can I use my own version of gcc and libg++?

Ptolemy 0.7 uses `gcc-2.7.2.2` and `libg++-2.7.2`, see “Dynamic linking fails” on page A-30 for information about Gnu versions and Dynamic linking.

To determine what version of gcc you are running, type `gcc -v`. To determine what version of libg++ you are running look at the libg++ filename.

```
cxh@kahn 32% gcc -v
Reading specs from /users/ptolemy/gnu/sol2.5/lib/gcc-lib/
sparc-sun-solaris2.5.1/2.7.2.2/specs
gcc version 2.7.2.2
cxh@kahn 33% ls /users/ptolemy/gnu/sol2.5/lib/libg++.so*
/users/ptolemy/gnu/sol2.5/lib/libg++.so@
/users/ptolemy/gnu/sol2.5/lib/libg++.so.2.7.2*
```

Ptolemy is configured to use shared library versions of `libg++` and `libstdc++` if they are supported on your platform. To compile Ptolemy from scratch, you should be sure that you have these libraries. Under Solaris, the libraries are named `libg++.so` and `libstdc++.so`. Under HPUX10.x, these libraries are named `libg++.sl` and `libstdc++.sl`.

A common problem is that the proper version of gcc is installed, but only the static libraries were built.

Under Solaris, you might see error messages like:

```
ld: fatal: relocations remain against allocatable but non-writable
```

The fix is to configure `gcc` and `libg++` with `--enable-shared` and build the shared libraries or to download the prebuilt Gnu binaries. See `ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/ptolemy0.7/html/g++shared.txt` for more information.

Can’t find genStarList or genStarTable during recompilation

The solution is to include `$PTOLEMY/bin` in your path. We don’t include certain files that are derived from other files. In the star directories, `.cc` and `.h` files are derived from `.pl` files, and the *domainname*stars.cc file is generated from `$PTOLEMY/bin/genStarTable`, .

“CGCMakefileTarget.h: No such file or directory” while linking pigiRpc

If you are in `$PTOLEMY/obj.$PTARCH/pigiRpc`, and you type `make`, and `$PTOLEMY/obj.$PTARCH/domains/cgc/targets/main/CGCMakefileTarget.o` does not exist, then `make` will try to create it. Unfortunately, `make` does not have the include files right, so `CGCMakefileTarget.h` is not found. There is nothing particularly special about `CGCMakefileTarget.o.` It is just first in the list of files on which `pigiRpc` depends.

The workaround is to run `make` from `$PTOLEMY`, rather than `$PTOLEMY/obj.$PTARCH.pigiRpc`.

Eventually, we would like to fix this so that it is not necessary to build in other directories before building in `$PTOLEMY/obj.$PTARCH/pigiRpc`. The solution here would be to move more `.o` files into `lib.$PTARCH`.

Missing symbols while linking pigiRpc

On the sun4 with libg++-2.5.2, if you compile pigiRpc with the g++ `-O` option, then you may have missing symbols while linking `pigiRpc`. The workaround is to upgrade to a newer version of libg++. If you are using prebuilt GNU binaries, then you may need to set some environment variables (see “Environment variables” on page A-24).

If, at link time, you see messages about undefined symbols, and the undefined symbols begin with `_vt`, then you probably have compiler version incompatibilities. In earlier releases of `libg++`, we have seen cases where the symbol `vt$7istream$3ios` is undefined. This symbol should be present in `libg++.a`, but it seems that if the compiler is not built with `-O2`, then this symbol will not be present in `libg++.a`. The workaround is to rebuild the library by hand, with something like:

```
cd obj.$PTARCH/gnu
make CC=/users/ptolemy/gnu/sun4/bin/gcc CXX=/users/ptolemy/gnu/sun4/bin/g++ CFLAGS=”-g -O2” CXXFLAGS=”-g -O2”
```

Note that you can find out what files have the undefined symbols by using the Unix `nm` command. For example on Suns, the command `nm -o $PTOLEMY/lib.sun4/* | grep MySymbol` will find all the files that have symbols that contain the string `MySymbol`. See

"Using c++filt to demangle symbols" on page A-26 for information about how to interpret symbol names in a library.

If, at link time, you see messages about undefined `ifstream` symbols in `libptolemy.a`, then the problem could be that you are using `gcc-2.7.2`, but linking against `gcc-2.7.2.2` libraries. Brian Evans pointed out that fix is to remove the `libg++` and `libstdc++` libraries in `$PTOLEMY/gnu/$PTARCH/lib` and create symbolic links to your local Gnu installation.

A.5.9 Dynamic linking fails

Ptolemy has the ability to load stars dynamically during run time. The stars are compiled into `.o` files and loaded with the Unix loader or with the `dlopen()` function. Dynamic linking is tricky and dependent on the Unix loader. There are several reasons dynamic linking can fail:

- If you are upgrading from an earlier release be sure to remove all the .o files in the directory where the source files for you star is located.
- If you are using prebuilt Ptolemy binaries, be sure that you are using the prebuilt GNU compiler that is also available with the Ptolemy binaries (Ptolemy0.7 was built with `gcc-2.7.2.2`/`libg++-2.7.2`) The alternative is to rebuild Ptolemy with your local GNU compiler, but be aware that versions earlier than `gcc-2.5.6` and `libg++-2.5.3` have bugs. `gcc-2.4.x` and `libg++-2.4.x` and earlier are known to have serious bugs, so you may want to upgrade. For more information, see "Gnu Installation" on page A-7.
- If you are using prebuilt Ptolemy binaries and have the prebuilt GNU compiler, be sure that either the Ptolemy distribution is available as `/users/ptolemy`, or you are setting the GNU environment variables in `$PTOLEMY/bin/g++-setup`. Again, see the GNU Installation section.
- If, at link time, you see messages about undefined symbols, then see the section above, "Missing symbols while linking pigiRpc".
- Dynamic linking may not work on machines using a different release of the operating system than that used to build the Ptolemy binaries. The solution is to rebuild Ptolemy from source.
- If you are having problems compiling the star from Ptolemy, try running the compile by hand from the shell. Unfortunately, part of the compile command is not always visible in the vem window. To see the all of the compile command, try running a few `vem` commands, such as 'i' (look-inside) to flush the vem buffer. Once you have produced a `.o` file, you can load the `.o` file into Ptolemy with the load-star command.
- If you cannot compile your star from the shell, try compiling a simple c++ program to verify that the compiler is working. Place the code below in a file called `hello.cc` and if you are using the GNU compiler, try compiling it with `g++ -v`. The `-v` option will show the compiler steps.

```
#include <stream.h>
main(){ cout << "Hello, Ptolemy.\n";}
```

- If you are having problems with undefined symbols at load time, try compiling your star with the same level of optimization as the binary was built with. We ship `pigiRpc` and `ptcl` binaries that have been compiled with `-O2`, so you may want to compile your star with `-O2`.
- If you are running under HPUX9.x, and you see messages like:

  ```
  collect2: ld returned 1 exit status
  /bin/ld: Invalid loader fixup needed
  ```

  then you probably need to create a `make.template` file to load your stars. The problem here is that Ptolemy attempts to compile your star with default arguments, however, since HPUX9.x uses `shl_load()` style linking, you need special compiler arguments, so you need a makefile See `$PTOLEMY/mk/userstars.mk` for more information.

  You may also need to upgrade your version of the GNU assembler named `gas`. Version 2.5.2 has been reported to have the problem, while version 2.6 seems to work fine. Note that `gas` is often named `as`. If you compile your star with the `g++ -v` option, then you will see which assembler the compiler is using. You can then call the assembler with the `--version` flag to see what version the assembler is.

A.5.10 Dynamic linking and makefiles

You may find it easier to use a `makefile` to build `.o` files for incremental linking. As part of the incremental linking process, `pigi` checks for the existence of a `Makefile` or `makefile` in the directory where the star resides. If a `Makefile` or `makefile` exists, then `make` *`XXXStarName`*`.o` is run, where *`XXXStarName`*`.o` is the name of the `.o` file to be incrementally loaded.

Another approach is to create a `make.template` file in the directory that contains rules to convert the `.pl` file to a `.o` file. If the `make.template` file includes `$PTOLEMY/mk/userstars.mk`, then most of the configuration is done. For example, to include a star `SDFSensorExcitation`, with optimization set at `-O2`, the `make.template` would contain:

```
ROOT = $(PTOLEMY)
VPATH = .
OPTIMIZATION=-O2
include $(ROOT)/mk/config-$(PTARCH).mk
INCL = -I$(ROOT)/src/domains/sdf/kernel -I$(KERNDIR)
PL_SRCS = SDFSensorExcitation.pl
DOMAIN = SDF
include $(ROOT)/mk/userstars.mk
```

Then, from a shell, the command to execute would be `make -f make.template depend` and then, from within `pigi`, it would be possible to link in the star. See the contents of `userstars.mk` for complete instructions.

If you have a star that requires multiple `.o` files, Tom Parks points out that `ld -r` might help. For example if `SDFWirelessChannel.o` uses functions from `Wireless.o`, the following commands might help:

```
ld -r SDFWirelessChannel.o Wireless.o
mv a.out SDFWirelessChannel.o
```

To load in multiple stars, you may find the `ptcl multilink` command useful.

A.5.11 Path and/or environment variables not set in "debug" pigi

When running Ptolemy's interactive graphical interface with the debug option

```
pigi -debug
```

the path may not be set correctly, or environment variables are not at their normal values. This is caused by the GNU debugger, `gdb`, overwriting values set by the `pigi` start-up script. From the `gdb` manual:

```
*Warning:* GDB runs your program using the shell indicated by your
`SHELL' environment variable if it exists (or `/bin/sh' if not). If
your `SHELL' variable names a shell that runs an initialization file-
-such as `.cshrc' for C-shell, or `.bashrc' for BASH--any variables
you set in that file affect your program. You may wish to move setting
of environment variables to files that are only run when you sign on,
such as `.login' or `.profile'.
```

If your `.cshrc` file specifies a value for a variable, it will override anything in the pigi start-up script. If this is the case, perhaps setting the `SHELL` environment variable to `/bin/sh` before firing off the debugger will fix the problem.

1.5.12 DE Performance Issues

DE Performance can be an issue with large, long-running universes. Below we discuss a few potential solutions.

Tom Lane pointed out that the Calendar Queue scheduler can be slower than the old DE scheduler if your time stamps span a wide range. This is because the Calendar Queue Scheduler tries to set up too many bins spanning the range. The old DE scheduler may work faster, as it keeps a queue of current-scheduled events, which is often fairly short.

Tom Lane also pointed out:

> If you have both a wide range of timestamps and a lot of future events in the queue at once, you might find it would help to improve the `PriorityQueue` code to provide a genuine priority queue (i.e., a heap, with O(log N) performance) rather than a simple list like it is now. But you ought to profile first to see if that's really a time sink.
>
> Also, you have to keep in mind that the overhead for selecting a next event and firing a star is not trivial. It helps if your stars do a reasonable amount of useful work per firing.

A few other points that may help you:

DE simulation can have certain inherently high cost, so using SDF or DE with SDF functionality inside wormholes can greatly improve performance.

- If you are running a long simulation, you should be sure that your machine is not paging or worse yet swapping, you should have plenty of memory. Usually 64Mb is enough, though 128Mb can help. Depending on what platform you are on, you may be able to use the program `top` (ftp://eecs.nwu.edu/pub/top). You might also find it useful to use `iostat` to see if you are paging or swapping.

- One way to gain a slight amount of speed is to avoid the GUI interface entirely by using `ptcl`, which does not have Tk or Higher Order Function (HOF) stars. See "Some hints on advanced uses of ptcl with pigi" on page 3-19 for details.

A.6 Known bugs

There are a number of known bugs that we have not had a chance to fix. You may find useful information about architecture dependencies in `$PTOLEMY/mk/config-$PTARCH.mk` for the particular architecture you are running under.

A.6.1 Bugs in vem

- There is no interlock between commands started by Vem and commands started by pigiRpc. If either process requests the other to perform some operation while the other process is trying to request the first one to do something, they end up in a "deadly embrace". The usual result is that pigiRpc exits with a complaint about a protocol error. (Fortunately, Vem continues to run, so you can at least save your schematics before restarting.) Examples of the problem include trying to issue edit-parameters or look-inside commands (which are invoked in a Vem window but require cooperation from pigiRpc) while pigiRpc is compiling a facet or running a simulation with graphical animation turned on (which require Vem to react to requests from pigiRpc). The simplest rule that will keep you out of trouble is "don't do anything in Vem windows while pigiRpc is busy".
- Deleting wires that are too long (i.e., end of wire is past the terminal, but near it) can result in `vem` dumping core.
- Labels that end with a carriage return are neither displayed nor printed correctly.
- Editing icons stimulates a memory leak in `vem` that can make the process grow quite big. After editing a few dozen icons, you may wish to exit `vem` and restart.
- The "layer" command only works when editing icons, not when editing schematics.
- The "copy-objects" command will copy from one window to another only when editing icons, not when editing schematics.
- `vem` or `pigiRpc` may fail to start up on 4 bit (16 color) screens. At UC Berkeley, we use 8 bit and 24 bit screens. Black and white (1 bit) support in vem is a little weak. If you are on a black and white screen, you may need to modify `$PTOLEMY/lib/pigiXRes9`. (Note that `$PTOLEMY/lib/pigiXRes9.bw` is read when the `-bw` `pigi` option is used to create black and white screen dumps from a color monitor. `pigiXRes9.bw` has very little to do with running on a black and white screen).
- If the `VEMBINARY` environment variable is set, then `bin/pigiEnv.csh` will use the contents of that variable as the `vem` binary. However, the binary must be named `vem`, or there will be minor problems with X resources, such as small fonts and incorrect snap. Most users will never use the `VEMBINARY` environment variable. It is primarily

used to debug `vem`.

- If your X server runs out of colors, `vem` may crash. One workaround is to not run color hogs like `Netscape`, or to limit the number of colors `Netscape` can use.
- Bug fixes to `vem` now mean that `vem` is more picky about text label heights. Currently, the range for text label heights is 0 to 80. If you select text and then use E, to edit the label, you may see an error if your text height is greater than 80. One workaround is to delete the text and reenter it with a height that is within the proper range. Another workaround is to use an old `vem` binary to edit the height and set it to within the range.
- If you re-read a facet that has an open run window, you won't be able to dismiss the run window. The workaround is to open a new run window.

A.6.2 Bugs in pigi

- It is possible to make more than one icon to represent a given star. For instance, the icon `And` and `And.input=2` refer to the same star, but the former can have any number of inputs, while the latter has exactly two inputs. As of this writing, this facility does not work for galaxies. You should create only one icon for each galaxy.
- `$PTOLEMY/bin/pigi` is a link to `$PTOLEMY/bin/pigiEnv.csh` which may add to your path. Under certain circumstances, you may get the message 'Warning: ridiculously long PATH truncated'. To work around this problem, try shortening your path. One trick is to use shorter pathnames in your path, perhaps using symbolic links.
- If your X server runs out of colors, `pigi` may crash. One workaround is to not run color hogs like `Netscape`, or to limit the number of colors `Netscape` can use.
- The compile-SDF target fails if a galaxy has incrementally linked in stars. The problem is that the `.h` file for the incrementally linked in star cannot be found by compile-SDF. As a workaround try editing the makefile in `~/PTOLEMY_SYSTEMS` and adding the directory where the incrementally linked stars are. Then build the binary by hand.
- The compile-SDF target cannot handle star parameters that use the Tcl Interpreter to evaluate Tcl commands.
- On some machines, the SDF Matlab demonstrations will appear to hang Ptolemy. This is a known bug in the Matlab external interface library that requires that the Matlab process be attached to a terminal. There are two workarounds: (1) run Ptolemy in the foreground, or (2) manually control the Ptolemy interface to Matlab by using the `matlab` command in `ptcl` (see 3.9.10 on page 3-16) via the Ptolemy console. In both workarounds, the Matlab process will be attached to a terminal.
- Ptolemy0.7 will not work with Matlab5 or Mathematica3. We are working on upgrading the interface. In the short term, you must use Matlab4 or Mathematica2.2.
- The Networks Of Workstations active messages (`NOWam`) CGC target is slow, but it does run under Solaris. However these demos will not work in an environment that requires Kerberos for `rsh`-ing jobs.
- To use the Dynamic Dataflow Code Generation (`cgddf`) CGC target, a file must be recompiled. The changes to `parScheduler.cc` are a fairly radical shift in how

Ptolemy handles disconnected graphs. These changes could break other parallel schedulers. The script `$PTOLEMY/bin/mkcgddf` will build a `pigiRpc.ptrim.debug` binary that contains the `cgddf` target. This script is basically untested, but should work.

- Mixed CGC/VHDL/TclTk demos leave `ptvhdlsim` running after exiting.
- Tom Lane pointed out the following problem in 0.7:

  > An initializable delay attached to any multiporthole output will fail. For example "Fork -> Printer", if you use the double-arrowed form of the Fork icon and put an initializable delay on the arc.

A.6.3 Bugs in tycho

Development of `tycho` is ongoing, so the version included in Ptolemy 0.7 is by no means the final version. In particular, there are many more features to be implemented.

- If you start up tycho with Ptolemy in the background (i.e, `tycho -ptrim &`), then exiting from the Matlab console may hang. The workaround is to place your `tycho` process into the foreground. This is a known bug with Matlab.
- See `$PTOLEMY/tycho/doc/bugs.html` for a more complete list of Tycho bugs.

A.6.4 Code generation bugs

- The CGC multirate `filterbank` and CGC `chaoticBits` demos bring up messages about initializable delays.
- If you try to run the CGC demos that generate sound, and you do not have a properly configured audio device, or you do not have write permission to `/dev/audio`, then you get a cryptic and uninformative error message, rather than something useful. The CGC `tremolo` demo will probably only work on a Sun SPARCstation running SunOS4.1.3 or Solaris2.x. The CGC `alive` demo will only work on an SGI.
- The use of initializable delays and variable delays in the code generation domains may result in incorrect code being generated. Variable delays are delays which depend upon the value of a galaxy or universe parameter. The workaround is to use regular delays of a constant value (i.e. "4") in the code generation domains. Note that initializable and variable delays work fine in the simulation domains.

  For example, the `CGC:multirate:filterbank` demo produces initializable delay warnings. The output from the `CGC:multirate:filterbank` demo is different than the output from the `SDF:multirate:filterbank` demo in that the original and reconstructed signals don't overlap as much in the CGC version.

A.6.5 Bugs in `pxgraph`

- If `pxgraph` is given exceptional numeric input, such as the IEEE floating point `Inf`, `-Inf`, or `NaN`, then it displays a blank window only.
- If `pxgraph` is given an empty file to plot, all it does is send a message "problems with input data" to the standard output.

- There is no way to produce hardcopy without running `pxgraph` interactively.
- Specifying the colors for data sets using X resources does not work.

A.6.6 HPPA specific bugs

- If you are rebuilding Ptolemy on the hppa, you must have X11 installed. Apparently, HP ships machines without most of the X11 include files. The prebuilt binaries also use libraries from X11R6. See `ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/contrib/hpux`.
- If you are building the GNU tools under HPUX10.20, you will also need GNU sed, which is downloadable from the Ptolemy ftp site in `ftp://ptolemy.eecs.berkeley.edu/pub/gnu/hppa` and you may need to patch HPUX. See `$PTOLEMY/src/gnu/README` for details.
- If you are using the prebuilt binaries under HPUX10.20, then you will need to install X11R6 shared libraries. See `ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/ptolemy0.7/README.hpux`
- The CGC demo `animatedLMS` demo dumps core upon start-up.
- On the hppa, under the HP C++ compiler, `Xhistogram` may have rounding problems.
- On the hppa, the CGC `animatedLMS` demo requires editing of Target Parameters to compile.
- On the hppa, the CGC `animatedLMS` demo gets an Arithmetic Exception and core dumps.
- On the hppa, the SDF `animatedLMS` demo generates a “`non-numeric value`” error in Tcl if the step size is so large that the system is made to go unstable.
- Compilation under older versions of `hppa.cfront` may fail. The problem seems to be with the `+A ld` command line argument. You must use the `+A ld` argument when compiling `pigiRpc` and `ptcl`, or incremental linking of new stars will fail. However, use of the `+A` argument produces the warning:

  ```
  CC: error: could not find __head symbol. You must use CC to
  link. If your main is not in C++, a call to _main() is
  required.(740)
  ```

  Even if this warning is printed, a viable binary is still produced. The `+A ld` argument is not necessary for other binaries, such as `vem` and `pxgraph`.
- If you are running an older version of the HP Cfront compiler, you may need to add `-DPOSTFIX_OPT=` to your c++ command line. See `config-hppa.cfront.mk`.
- If you are having problems incrementally linking in stars, and you are getting messages like:
  `/bin/ld: (Warning) Inter-quadrant branch in XXX`
  where `XXX` is the name of the `.o` file you are trying to link in,
  then you may need to be compiling your stars with the proper level of optimization.

See "Dynamic linking and makefiles" on page A-31 for more information.

- If you have problems with `ld` under HPUX, you should try patching your operating system with a patch from HP. Try `http://europe-support.external.hp.com` if you are in Europe, `http://us-support.external.hp.com` if you are anywhere else.
- If, under HPUX9.x, you get a message like:
  `ld: fatal: relocations remain against allocatable but non-writable sections`
  Then you may need to apply a patch to `ld`.
- To incrementally link new stars under HPUX9.x, you will probably need to supply a make.template file. See "Dynamic linking fails" on page A-30.
- The `multilink` command seems to trigger `Inter-quadrant branch messages`, and then hang pigi. We are not sure why.
- HPUX10.x will build a `pigiRpc` with the Process Network (PN) domain, but the event loop is broken. The PN domain is not part of the default hppa build, so this problem will not affect most users.
- If you are under HPUX10, then building the PN domain requires Distributed Computing Environment (DCE) threads. You will need to install the DCE development set of the OS CDs. If you don't have a `/usr/include/pthread.h`, then you probably don't have the DCE development set installed.

A.6.7 IBM AIX specific bugs

Xavier Warzee suggest the following for gcc and AIX.

To generate `gcc-2.7.2` under AIX3.2.5 with `cc`, you need the PTF U436313. The PTF U432238 is suggested in the README.RS6000 file from the `gcc-2.7.2` distribution, but this PTF is superseded by the PTF U436313. This PTF allows you to upgrade the cc IBM compiler from the release 1.3.0.0 (delivered with AIX 3.2.5) to the release 1.3.0.33 which compiles gcc-2.3.6.

A.6.8 Silicon Graphics IRIX5 specific bugs

The Silicon Graphics port is not one of our main ports, so there are several serious bugs:

- Linking of a full size `pigiRpc` binary can fail with a 'GOT Overflow' message. The problem is that `pigiRpc` has too many symbols. The Irix `dso` man page suggests using shared libraries, but `gcc-2.7.2` may not support C++ shared libraries under Irix. The workaround is to run `ptrimRpc` rather than `pigiRpc`.
- Installation of the `xv` man pages will fail. Irix does not have `nroff` and `tbl`.
- To build the CGC demos, you may need to exit `pigi`, set the environment variable `SGI_CC` to `-cckr`:
  `setenv SGI_CC -cckr`
  and restart `pigi`. See the sgi `cc` man page for more information.

- `sdf:image:motionCompensation` demo has some odd looking blocks in the McompOutput.1 image. Some of the images from the `irix5` run of this demo do not look like images in the sun4 version of this demo.

A.6.9 Linux specific bugs

We do not have access to a Linux system onsite at UC Berkeley, so our support of Linux is very limited. If you are having problems with prebuilt Linux binaries, you may want to try rebuilding Ptolemy from scratch. You may also want to check the ptolemy-hackers archives, located at the bottom of the Ptolemy home page at `http://ptolemy.eecs.berkeley.edu`.

- Linux uses GNU `ar`, which seems to have problems if it is run on two files whose names are not unique in the first 13 characters. We have renamed a number of stars to workaround this problem. The `make checkjunk` command will report the names of files that are not unique in the first 13 characters. You should only have problems with this if you are creating stars of your own and placing them in libraries.
- If you are running the Linux version on a standalone machine, then please refer to section “Mr. Ptolemy window does not come up” on page A-16.
- Richard Nicholls <R.Nicholls@mmu.ac.uk> reports that the `ld` with GNU binutils 2.5.2 does not support dynamic loading (See the `TODO` file distributed with the GNU `binutils` source). One solution is to use an older version of GNU `ld`. Another solution would be to use `dlopen()` style incremental linking, see `$PTOLEMY/src/kernel/Linker.sysdep.h`.

A.6.10 Sun Solaris 2.4 specific bugs

- If you are building gcc under Solaris, you must have `/bin` in your path before `/usr/ucb` see `$PTOLEMY/src/gnu/README`.
- If, while linking, you get a message like
  ```
  unable to locate archive symbol table: Format error: archive fmag
  ```
  then you have probably run out of swap space. See the Solaris `swap` command for how to add more swap with `mkfile`.
- Under Solaris 2.5 with SunSoft CC 4.1, `src/kernel/MatrixParticle.cc` fails to compile. The fix for this may require substantial reimplementation of the Ptolemy `Matrix` class, so the changes did not make it into the 0.7 release.
- The Utah Raster Toolkit does not come with a Solaris2.4 configuration file. We are running SunOS binaries in-house. See “Other useful software packages” on page A-14 for more information.
- The GNU make binary built for Solaris2.5 will not work with Solaris2.4. If you try to run the Solaris2.5 GNU make binary under Solaris2.4, you will get an error message like:
  ```
  ld.so.1: make: fatal: relocation error: symbol not found: setlinebuf: referenced in make
  ```

A.6.11 Sun OS4 specific bugs

- If you are using the prebuilt GNU binaries in `pt-0.7.gnu.sun4.tar.gz`, then you may have problems if your Ptolemy distribution is not at /users/ptolemy, and you try setting the GNU environment variables with the `$PTOLEMY/bin/g++-setup script`. We are working towards a solution, but you might find the following variables will work for you:
  ```
  unsetenv GCC_EXEC_PREFIX
  unsetenv C_INCLUDE_PATH
  unsetenv CPLUS_INCLUDE_PATH
  setenv LIBRARY_PATH $PTOLEMY/gnu/$PTARCH/lib
  setenv COMPILER_PATH $PTOLEMY/gnu/$PTARCH/lib/gcc-lib/$PTARCH/
  2.7.2.2
  setenv GCC_INCLUDE_DIR $PTOLEMY/gnu/$PTARCH/lib/gcc-lib
  ```
  We are working on a solution for this.
- Under SunOS4.x, we use the older BSD `ld` style incremental linking because of problems with g++ shared libraries. This means that the SunOS4.x binaries must be linked statically instead of dynamically, so the binaries are much larger than on platforms that use `dlopen()` style incremental linking. It also means that only the symbols that are used by stars present at link time are actually present in the binary. This can be a problem if your incrementally linked star uses a symbol from `libg++.a` that is not used by a star present at link time. See Appendix D, "Shared Libraries" for more information.

A.6.12 DEC Alpha specific bugs

- The DEC Alpha port is a new port that has had very little testing, so there are bound to be 64bit vs. 32bit bugs
- The CGC fixed-point demos give incorrect results. This is most likely due to the fact that the DEC Alpha is 64 bits.
- SDF/DDF Wormholes cause `SIGFPE` signals, which crash `pigi`. The DDF `ifThenElse` demo uses such SDF/DDF Wormhole.
- The full pigiRpc may fail to start with messages like:
  ```
  /sbin/loader: Fatal Error: lazy_text_resolve: symbol malloc
  should not have any relocation entry
  ```
  The workaround is to do:
  ```
  setenv LD_BIND_NOW yes
  ```
  See the DEC Unix `loader` man page for more information about `LD_BIND_NOW`. You may see similar error messages when you run the CGC fixed point demos.
- Vem produces lots of `Unaligned Access` messages. You can run
  ```
  uac p noprint
  ```
  to turn them off in the current shell.

A.6.13 GNU compiler bugs

- If you write your own stars, then be aware that `gcc-2.7.2` may not choose the expected cast. In particular, `g++` has been known to cast everything to `Fix` if the explicit cast is omitted. The arithmetic is then performed using fixed-point computations. This will be dramatically slower than double or integer arithmetic, and may yield unexpected results. It is best to explicitly cast states to the desired form. For more information, see the ptlang chapter in the Ptolemy Programmers Manual.

- If you already have GNU make installed, and the binary is called `gmake`, then the installation of the Ptolemy GNU tools may fail. The workaround is to create a link in your path from `make` to `gmake`.

A.7 Additional resources

The best, most complete source for information on Ptolemy is to be found in the Ptolemy manual, *The Almagest*. The manual is included in every distribution and is available in our WWW and FTP sites.

A second source is the `ptolemy-hackers@ptolemy.eecs.berkeley.edu` mailing list. This list provides a forum for Ptolemy questions and issues. Users of the current release who have a Ptolemy question, comment, or think they've found a bug should send mail to `ptolemy-hackers`. Archives of the mailing list are also available. See the "Ptolemy mailing lists and the Ptolemy newsgroup" on page A-2.

A third source is the Ptolemy Usenet news group `comp.soft-sys.ptolemy`.

A fourth source on the latest information about Ptolemy is the Ptolemy World Wide Web (WWW) server accessible by programs such as Netscape. The Universal Resource Locator (URL) for the Ptolemy WWW server is `http://ptolemy.eecs.berkeley.edu`. The WWW pages contain on-line access to a quick tour of Ptolemy, a list of publications, a collection of technical papers, information on the members of the development team, and links to WWW information on related tools. The WWW interface is a superset of our FTP server discussed below.

Another source is the Ptolemy FTP site. To access this, use anonymous FTP to `ptolemy.eecs.berkeley.edu.` The directory `pub/ptolemy/papers` contains some of the latest Ptolemy papers and articles.

A.8 Submitting a bug report

Ptolemy is offered without support, but we are nonetheless interested in hearing your feedback with regard to bugs. A good bug report consists of the following elements:

- What OS you are running under (e.g. SunOS4.1.3, Solaris2.5.1, HPUX-10.01).
- What version of Ptolemy you are running. If you are not running the latest version, you may want to upgrade and see if your bug has been fixed already.
- Whether you are using prebuilt binaries, or if you compiled your binaries yourself.
- If the problem is with the compiler, and you are using the Gnu compiler, you should also state whether you are using the prebuilt compiler we provide, or your own version. You should also try running `gcc -v` to see what version of the compiler you are running.
- It is best for us if you can reproduce the bug in a small test case and send us your `~/`

`pigiLog.pt` file, along with a detailed description of what you did and what happened.

- The best place to mail a bug report is to send it to the Ptolemy-hackers mailing list at `ptolemy-hackers@ptolemy.eecs.berkeley.edu`.
- If the bug consists of a problem with facets, you might want to uuencoded these facets and include them in your mail message. Note that there is a 40K size limit to mail to Ptolemy-hackers, so if your mail message is large, then you may want to send it only to `ptolemy@ptolemy.eecs.berkeley.edu`. To uuencode a facet and a galaxy:

```
tar -cf /tmp/facets.tar yourfacet yourgalaxy
uuencode /tmp/facets.tar facets.tar > facets.uu
```

- Then include the `facets.uu` file in a mail message. You may get some leverage out of compressing the `facets.tar` file before uuencoding it.

Appendix B. Introduction to the X Window System

Pigi uses the X window system, version 11 release 6 (X11R6). It will also run correctly under X11R4, X11R5 and under Sun OpenWindows. This appendix provides just enough information about using X so that if you are not familiar with it you will nonetheless be able to get started. Complete documentation, however would be helpful. Complete guides to the X window system are available in many technical bookstores. If you are familiar with X, and your user account is properly configured to run X11, you need not be concerned about this appendix. `Pigi` will automatically load the resources it needs for operation. If you wish to customize these resources, it may be useful to scan this section. Resources are explained below.

B.1 A model user's home directory

A strength (and weakness) of the X window system is that it is infinitely configurable. Making reasonable use of it requires a number of files in the user's home directory. Usually, these files are hidden by giving them a name that begins with a period (`.`), such as `.xinitrc`. You can see these files by listing your directory contents with the `-a` option:

```
ls -a
```

Even if you are new to Unix systems, with a new account, your system administrator will have put a number of such hidden files in your home directory when he or she sets up your account. Hopefully, these files have been chosen to give you reasonable behavior immediately. If not, however, this appendix can help you set up your account for running Ptolemy.

The home directory of your Ptolemy installation (either `~ptolemy` or `$PTOLEMY`) is designed to serve as the home directory of a model user. We will refer to the directory by the name `$PTOLEMY`. Here is a listing of the contents of this directory:

```
% ls -a
./                .cshrc            .twmrc
../               .login            .xsession
.Xdefaults        .plan
```

This user is configured according to the tastes of the authors of Ptolemy. To configure your own account to behave the same way, you can copy all the dot files (hidden files whose names begin with a period) from the directory `$PTOLEMY` into your home directory. Unless you are a new user, however, you will probably not want to overwrite features already defined in your own dot files. In this case, you will need to extract the features you desire from the dot files in `$PTOLEMY` and add them to your own. To make this easy to do, we explain below the salient features of each of these files.

B.2 Running Pigi using Sun's OpenWindows system

Pigi will run under Sun's OpenWindows Version 2 or 3, assuming the Athena widgets have been installed (see the installation instructions). We assume as a starting point that you are already configured to run OpenWindows; *do* not attempt to use the `.login` and `.cshrc` in the ptolemy account in this case. First, make sure that your .cshrc file includes `$PTOLEMY/bin` in your path.

You will need to have the `olwm` window manager run in the mode where focus follows the mouse (in this mode, your keystrokes go to whatever window your mouse points to); to do this, find the line in your `.xinitrc` file that invokes `olwm`; add the option `-follow` to the `olwm` command. If, for some reason, you must run `olwm` in the click-to-focus mode, you must click the mouse in each window before pigi will accept keyboard commands in that window; it is possible but annoying to use pigi this way. An alternative is to add the following line to your `.Xdefaults` file:

```
        OpenWindows.SetInput:followmouse
```

B.3 Starting X

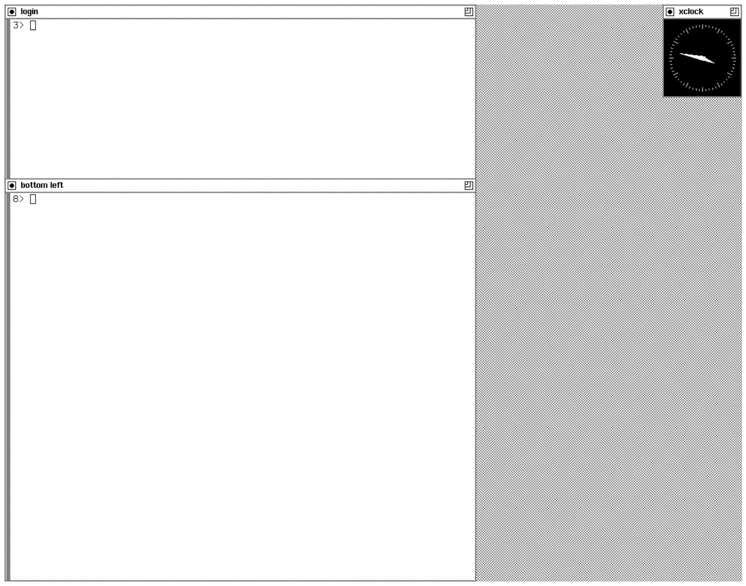
There are two principal ways that X may be configured:

- The X server may be running all the time. In this configuration, when no one is logged in a single window together with a background appears. You don’t have to worry about starting X. A daemon program called *xdm* is responsible for logging you in.
- The user may be responsible for starting X — in this case, when you log in you get a bare workstation tube.

Our sample configuration attempts to support both to some degree. If `xdm` is running, it will use the supplied `.xsession` file to set up windows. If `xdm` is not running, and the user is on a bare Sun tube, the `xinit` program will execute, starting the X server; the `.xinitrc` file is used in this case to configure windows. The supplied `.xinitrc` file reads:

```
        xrdb $HOME/.Xresources
        twm&
        xterm -geometry 80x30+0-0 -fn 9x15 -name "bottom left" &
        xclock -geometry 120x120-0+0 &
        exec xterm -geometry +0+0 -fn 9x15 -name "login" -ls
```

The first line reads X resources from the file `~/.Xresources`. Then the window manager called `twm` is started. The next three lines open windows using the `xterm` and `xclock` commands. The `.xsession` file works pretty much the same way for xdm installations. The screen will look something like this after start up:



The function of the mouse buttons after the windows have opened is determined by the `$PTOLEMY/.twmrc` file. This file is reasonably easy to interpret, even without any X experience, so you may wish to examine it and customize it. If your installation uses the X window system in some way that differs from our default, then whoever installed Ptolemy should have modified the `.login` file above to reflect this.

B.4 Manipulating Windows

If you already know how to use the X Window system and you are using your own window manager configuration rather than the standard Ptolemy configuration, you may skip this section. Assuming you use the file `$PTOLEMY/.twmrc` without modification, rather than your own window manager, the basic window manipulations are explained below. Note that there are often several different ways to accomplish the same objective. Feel free to experiment with key and mouse button combinations in various parts of the screen. First, you must identify the “meta” key on your keyboard. It may be labeled in any of various ways, including “meta”, “left”, “right”, “alt”, a small diamond, or any of a number of other possibilities. It is usually close to the shift and control keys. Most window manipulations require that you hold down the meta key while depressing mouse buttons.

Iconifying windows.

Depressing the meta key, and clicking the left mouse button in any window will iconify it. The window disappears, replaced by a symbol in the icon manager at the right of the screen. To get the window back, place the cursor in the appropriate slot of the icon manager, and click the left mouse button. An alternative iconifying mechanism is to click any mouse button on the icon symbol at the left of the window header.

Moving windows.

Holding the meta key and dragging with the middle mouse button will move a window. “Dragging” simply means to hold the mouse button while moving the mouse. Alternatively, you can drag the left button in the middle of the window header.

Resizing windows.

The meta key and right mouse button can be used to resize a window. Place the mouse cursor near a corner of a window, depress the meta key, and drag the right button. Without the meta key, any button in the rightmost corner of the window header will resize the window.

Mnemonic.

The window header has an iconify icon at the left, a blue bar for moving the window in the middle, and a resize symbol at the right. Correspondingly, without going to the window header, but using the meta key, the left mouse button will iconify a window, the middle button will move it, and the right button will resize it. Hence, the window header can be used as a mnemonic to help remember which mouse button has which function when the meta key is depressed.

Pick up and stuff.

In a text window, without the meta key, the mouse may be used to grab text and put it somewhere else, either in the same window, or in some other windows. This can be very useful to avoid copying long sections of text. The left mouse button, when depressed and dragged, highlights a region of text in a window, and stores the text. The right button can be used to modify the extent of the highlighted region. The highlighted text is "picked up", stored for future use. The middle button causes the highlighted text to be typed to whatever window has the mouse cursor.

New windows.

A useful command, defined in `$PTOLEMY/.cshrc`, is “term”. Typing this in any window, followed by a carriage return, opens a new terminal window with a reasonable size and color. You may change the size and location in the usual way.

Removing windows.

Most windows, including all pigi windows, can be removed by typing control-D with the mouse cursor in the window.

Resources.

A *resource* in X is a parameter that customizes the behavior of applications that run under X. For instance, resources determine the foreground and background color of your windows. Pigi requires that some resources be set. These resources are defined in the file `$PTOLEMY/lib/pigiXRes`, and are automatically merged with whatever other resources you may have defined. The merging occurs when you invoke the start-up script `$PTOLEMY/bin/pigi`. In addition to these required resources, there are many optional resources. Some of these are set in `$PTOLEMY/.Xresources`.

If you have not used the X window system before, you will probably not have a file with the name `.Xresources` in your home directory, and can simply copy the one from `$PTOLEMY`. This file defines some basic resources. If you already have a file with this name, then you can probably use the one you have as is.

Appendix C. Filter design programs

C.1 Introduction

Pending the inclusion of more sophisticated filter design software with Ptolemy, this distribution includes two C programs for this purpose. These design FIR filters using the Parks-McClellan algorithm or the window method. The frequency sampling method can be done directly using Ptolemy stars, as shown in the demos included with the SDF domain.

These programs can be invoked standalone or through the filter command in pigi. If invoked through `pigi`, then an `xterm` window will be opened and the program started.

Filter specifications can be entered by hand or loaded from a file. The first prompt from the program is the name of the input command file. Simply typing return will result in manual entry. To enter data from a file, the data should appear in the file in exactly the order that it would appear if it were being entered by hand, with one question answered per line. Unfortunately, these command files are rather difficult to read, and not too easy to create correctly. It is recommended to first do manual entry, then imitate the entries in a file. Data can be entered in any reasonable numeric format.

Caveats

These are public domain Fortran programs that have been converted to C, provided for convenience; they are not an integral part of Ptolemy. Incorrect formatting of data line can lead to ungraceful exits (often with a core dump) or incorrect results. Either program can be invoked through the Ptolemy graphical interface, in which case it is started in its own window, in the background. Note that in this case the program always starts in the current working directory of `pigi`, probably your home directory.

C.2 optfir — equiripple FIR filter design

`Optfir` is a Fortran program performing classical FIR filter design using the Parks-McClellan algorithm. There is nothing unusual about this program, so almost any DSP book will give an adequate explanation of the algorithm as well as the meaning of its arguments. Briefly, the program permits the design of bandpass filters, differentiators, hilbert transformers, and half-band filters.

Bandpass filters include lowpass, highpass, bandstop, and multiband, where each band can have a different gain (the "desired value"). A desired value of 0 specifies a stopband. The weight associated with each band determines the ratio of ripple in each band; a higher weight means less ripple.

The following example should serve to illustrate use of the program. Invoke the program through `pigi` by calling up “equiripple FIR” in the “Filter” menu. Alternatively, you can start the program directly from any terminal window by typing the command `optfir`. The questions posed by the program are indicated below. The text in **`Courier-Bold`** are your response.

```
Enter name of input command file (press <Enter> for manual
entry, Sorry, no tilde-expansion. Give path relative to your
home or start-up directory):
     <Return>
```

You can put your responses in a file to avoid having to type them each time you execute the program. Here, I assume you are typing them interactively.

```
Enter filter type (1=Bandpass, 2=Differentiator, 3=Hilbert
transformer, 4=Half-band):
     1
```

Lowpass, Highpass and Bandpass filters are all called Bandpass.

```
Enter filter length (enter 0 for estimate):
     32
```

The number of taps. If you enter 0, the program figures out how many you need for your specification.

```
Enter sampling rate of filter:
     1
```

Use 1Hz.

```
Enter number of filter bands:
     2
```

There will be a passband and a stopband.

```
Enter lower band edge for band 1:
     0
```

For a lowpass filter, this should be 0 for d.c.

```
Enter upper band edge for band 1:
     0.1
```

Upper edge of the passband.

```
Enter desired value for band 1:
     1
```

Specify unity gain in the passband.

```
Enter weight factor for band 1:
     1
```

Using 1 here and 1 in the stopband will give ripples of the same size in both bands. You could experiment with allowing more ripple in the passband by making this number smaller.

```
Enter lower band edge for band 2:
     0.15
```

Lower edge of the stopband.

```
Enter upper band edge for band 2:
     0.5
```

Since this is the Nyquist frequency, it specifies the top of the stopband.

```
Enter desired value for band 2:
     0
```

Zero here defines this to be the stopband.

```
        Enter weight factor for band 2:
             1
```

See comment above on weight.

```
        Do you want x/sin(x) predistortion? (y/n):
             n
```

Note that the program will design a filter that predistorts to compensate for the effect of the zero-order hold in a D/A converter.

```
        Enter name of coefficient output file (Sorry, no tilde-
        expansion. Give path relative to your home directory):
             filter_taps
```

The name of the file in which to store the impulse response of the design. The resulting file can be used in an FIR star or a WaveForm star using the syntax `< filename` to read it. Note that the file will be stored in your home directory.

```
         Executing ...


         Finite Impulse Response (FIR)
         Linear Phase Digital Filter Design
         Remez Exchange Algorithm
         Bandpass Filter
         Filter length = 32


         Impulse Response Coefficients:
         h( 1) = 0.2199929E-02 = h( 32)
         h( 2) = -0.1615105E-01 = h( 31)
         h( 3) = -0.1167266E-01 = h( 30)
         h( 4) = -0.5506404E-02 = h( 29)
         h( 5) = 0.6444952E-02 = h( 28)
         h( 6) = 0.1864344E-01 = h( 27)
         h( 7) = 0.2227415E-01 = h( 26)
         h( 8) = 0.1105212E-01 = h( 25)
         h( 9) = -0.1328082E-01 = h( 24)
         h( 10) = -0.3894535E-01 = h( 23)
         h( 11) = -0.4806972E-01 = h( 22)
         h( 12) = -0.2521652E-01 = h( 21)
         h( 13) = 0.3334328E-01 = h( 20)
         h( 14) = 0.1151020E+00 = h( 19)
         h( 15) = 0.1945332E+00 = h( 18)
         h( 16) = 0.2434263E+00 = h( 17)


         Lower band edge: 0.0000000 0.1500000
         Upper band edge: 0.1000000 0.5000000
         Desired value: 1.0000000 0.0000000
         Weight factor: 1.0000000 1.0000000
```

```
Deviation: 0.0236464 0.0236464
Deviation in dB: 0.4108557 -32.5247154
Extremal frequencies:
0.0000000 0.0312500 0.0625000 0.0878906 0.1000000
0.1500000 0.1617188 0.1871094 0.2183594 0.2496094
0.2828125 0.3160156 0.3492188 0.3824219 0.4156250
0.4507813 0.4839844
```

In the above, we have designed a lowpass filter with 32 taps with the edge of the passband at 0.1 Hz and the edge of the stopband at 0.15.

C.3 wfir — window method FIR filter design

`Wfir` is a C program performing classical FIR filter design using the window method. There is nothing unusual about this program, so almost any DSP book will give an adequate explanation of the algorithm as well as the meaning of its arguments. Briefly, the program permits the design of lowpass, highpass, bandpass, and bandstop filters using any of a number of windows. The method is to first compute the impulse response of an ideal (brick wall) filter, and then window it with the selected window to make the impulse response finite.

The window method can also be implemented directly using Ptolemy block diagrams. See “FIR filter design” on page 5-78.

Appendix D. Shared Libraries

Authors: *Christopher Hylands*
Alain Girault

D.1 Introduction

Shared libraries are a facility that can provide many benefits to software but have a slight cost of additional complications. In this appendix we discuss the pros and cons of shared libraries. For further information about shared libraries, you should consult the programmer's documentation that comes with your operating system, such as the Unix `ld` manual page.

D.1.1 Static Libraries

A static library file is a file that consists of an archive of object files (`.o` files) collected into one file by the `ar` program. Static libraries usually end with `.a` (i.e., `libg++.a`). At link time, static libraries are searched for each global function or variable symbol. If the symbol is found then the code for that symbol is copied into the binary. In addition, any other symbols that were in the original `.o` file for the symbol in question are also copied into the binary. In this way, if we need a symbol that is dependent on other functions in the `.o` file in which it is defined, at link time we get the dependent functions. There are several important details about linking, such as the order of libraries, that should be discussed in your system documentation.

D.1.2 Shared Libraries

Most modern operating systems have shared libraries that can be linked in at runtime. SunOS4.x, Solaris2.x and HPUX all have shared libraries.

Shared libraries allow multiple programs to share a library on disk, rather than copying code into a binary, resulting in smaller binaries. Also shared libraries allow a binary to access all of the symbols in a shared library at runtime, even if a symbol was not needed at link time.

A shared library consists of an archive of object files (`.o` files) collected into one file by either the compiler or the linker. Usually, to create a shared library, the `.o` files must be compiled into Position Independent Code (*PIC*) by the compiler. The compiler usually has a special option to produce PIC code, under `gcc`/`g++`, the `-fPIC` option produces PIC code. Shared libraries have suffixes that are architecture dependent: under SunOS4.1 and Solaris, shared libraries end with `.so` (i.e., `libg++.so`); under HPUX, shared libraries end with `.sl` (i.e., `libg++.sl`).

In addition, shared libraries can also have versioning information included in the name. Shared library versioning is architecture dependent, but a versioned shared library name might look like `libg++.so.2.7.1`. Note that the version of a shared library can be encoded in the shared library in the `SONAME` feature of that library. Usually, the `SONAME` of a library is the same as the filename (i.e., the `SONAME` of `/users/ptolemy/gnu/sol2/lib/libg++.so.2.7.1` would be `libg++.so.2.7.1`). Interestingly, if you rename a shared library without changing the `SONAME` and then link against the renamed shared library, then at runtime the binary may report that it cannot find the proper library.

The constraint with shared libraries is that the binary be able to find the shared libraries at run time. Exactly how this is done is architecture dependent, but in general the runtime linker looks for special environment variable that contains pathnames for directories to be searched. Under SunOS4.1.x and Solaris2.x, this environment variable is named `LD_LIBRARY_PATH`. Under HPUX, the variable is named `SHLIB_PATH`. A binary can also have a list of pathnames to be searched encoded inside it. Usually this is called the `RPATH`. In general, asking the user to set the `LD_LIBRARY_PATH` or `SHLIB_PATH` is frowned upon. It is better if the binary has the proper `RPATH` set at link time.

D.1.3 Differences between static and shared libraries: Unresolved symbols

A library consists of `.o` files archived together. A `.o` file inside a library might contain symbols (functions, variables etc.) that are not used by your program.

At link time, a static library can have unresolved symbols in it, as long as you don't need the unresolved symbols, and you don't need any symbol that is in a `.o` file that contains an unresolved symbol. However, with shared libraries, you must resolve all the symbols at link time, even if you don't necessarily use the unresolved symbol.

As an example, say you have a program that uses a symbol from the `pigi` library (`$PTOLEMY/lib.$PTARCH/libpigi.*`), but does not use Octtools which is used by other files that make up the `pigi` library

If you are linking with a static library, you can have some unresolved symbols in the static library, as long as you don't reference the unresolved symbols. So, in our example, you could just link with the static `libpigi.a`.

If you are linking with a shared `libpigi`, you must resolve all the unresolved symbols. So, if you need a symbol from the `libpigi` library, then you must also include references to the Octtools libraries that pigilib uses, even though you are not using Octtools. So you would have to link in `liboct.so` and `libport.so` and the other Octtools libraries.

One positive benefit of this is that *all* the symbols in pigilib are available at run time, which makes incremental linking much easier, especially if we have a shared g++ library.

D.1.4 Differences between static and shared libraries: Pulling in stars

If you are using static libraries, then for a symbol to be present in the binary, you must explicitly reference that symbol at link time. When building Ptolemy with static libraries, each star directory contains a *xxx*`stars.c` file (where *xxx* is the domain name, an example file is `$PTOLEMY/src/domains/sdf/stars/sdfstars.c`) which gets compiled into *xxx*`stars.o`. At link time, the *xxx*`stars.o` file is included in the link command and the linker searches `lib`*xxx*`stars.a` for the symbols defined in *xxx*`stars.o`, and pulls in the rest of the star definition.

If you are using shared libraries, then all the symbols in the `lib`*xxx*`stars` file are present at runtime, so you need not include the *xxx*`stars.o` file at link time.

D.2 Shared library problems

- Start up time of a binary that uses shared libraries is increased. We believe that some of the increased startup time comes from running star constructors. We are working on modifying Ptolemy so that startup time of binaries that use shared libraries is decreased. See “Startup Time” below for more information.
- The time necessary to start up a debugger is sometimes increased. When the debugger starts up, it usually has to load all the shared libraries so that the debugger knows where to find symbols.
- Building is more complex. Unfortunately, shared libraries are very architecture dependent. Also, the commands and command line arguments differ between architectures. Finally, how different versions of the same shared library are handled, along with the shared library naming conventions also vary between architectures.
- You need to keep track of where the shared libraries are, either by using `RPATH` at link time or setting `LD_LIBRARY_PATH` or `SHLIB_PATH`. The problem is that if you are building a C Code Generation (CGC) application that uses shared libraries, then at runtime the user needs to either have the necessary shared libraries in their `LD_LIBRARY_PATH` or `SHLIB_PATH`, or the binary needs to have the `RPATH` to the shared libraries encoded into it. This can be done with an option of the linker. The various commands to do this are architecture dependent, and `Default-CGC` target usually fails. The `Makefile_C` target and the `TclTk_Target` which is derived from `Makefile_C` is much more likely to work with shared libraries.
- It could be the case that binaries that use shared libraries might use slightly more memory.

D.2.1 Startup Time

In Ptolemy you can build a `pigiRpc` that has only the domains you are interested in with Jose Pino’s `mkPtolemyTree` script in `$PTOLEMY/bin`, see the Programmer’s Manual for more information. The startup time for a full `pigiRpc` is greater than for a `pigiRpc.ptrim` (SDF, DE, CGC and a few other small domains). If you use either `pigiRpc.ptrim` or `pigiRpc.ptiny` (SDF and DE only), then the start up time is quite reasonable. If you regularly use some of the other less common domains, then you can build special `pigiRpc` with just your domains.

One reason that startup time is increased might be because Ptolemy constructs a lot of objects and processes many lists of things like domains. We may be able to decrease startup time by carefully managing the star constructors.

One way to speed things up might be to create a large shared library that has the domains in which you are interested. Startup time might be faster if everything is in one file. Currently we have about 80 different shared libraries.

Combining these libraries into a few big libraries for ptiny, ptrim and pigi binaries might help. Of course, we could leave the 80 libraries and just add the new libraries. We have not tried this, but it might be interesting.

D.3 Reasons to use shared libraries

- All the symbols in a shared library are available at runtime. This is especially important with incremental linking of stars. If you have a shared `libg++`, then you can use all the symbols in `libg++` in a star for which you did not use the `libg++` symbol at link time. If you use static linking, then when you incrementally link, you only have symbols that you used when you linked the binary. The same is true for symbols in the Ptolemy kernel and the domain kernels.
- You don't need to write dummy functions to pull in code from a library. `$PTOLEMY/src/domains/sdf/stars/sdfstars.c` is an automatically generated C file that pulls in the stars from `libsdfstars.a`. If you use shared libraries, then you need not have a `sdfstars.c` file. Also, more than one person has been confused because they added new file containing new functionality to the Ptolemy kernel, and then when they tried to link in a star, the symbols they just wrote couldn't be found. Usually this is because they are not using the new symbols anywhere at link time, so the new symbols are not being pulled into the binary. If the Ptolemy kernel is a shared library, then this problem goes away, as the new symbols are present at incremental link time.
- Smaller binary size on disk. Shared library binaries are smaller on disk, so it is possible to have many versions of `pigiRpc` that include different domains, without using up a lot of disk space. If you use shared libraries, a `pigiRpc` is about 1.5Mb; if you use static libraries, then a `pigiRpc` is about 8Mb.
- Link time is greatly decreased with shared libraries. Under Solaris with shared libraries, it takes almost no time to link a binary. Under SunOS with static libraries it can take 8 minutes to link. Using a tool like Pure Inc.'s `purelink` can help, but `purelink` is expensive and is not available everywhere.
- If you are running multiple `pigis` on one machine, the memory usage should be reduced because of all the pigi binaries are sharing libraries. In theory, if a binary is built with static libraries, you should get some sharing of memory, but often the shared libraries result in better memory usage. If you use shared libraries for X11 and Tcl/Tk, then your memory usage should be lower.
- Using `dlopen()` to incrementally link in new stars is usually faster than the older method of using `ld -A`. Eventually, we may be able to link in entire domains at runtime using `dlopen()`.

D.4 Architectural Dependencies

In this section, we discuss shared library architectural dependencies

Table 1: Commands to use to find out information about a binary or library

Architecture	Command(s) that prints what libraries a binary needs	Library Path Environment Variable
`hppa`	`chatr` *file*	`SHLIB_PATH`
`irix5`	`elfdump -Dl` *file*	`LD_LIBRARY_PATH`

Architecture	Command(s) that prints what libraries a binary needs	Library Path Environment Variable
`sol2`	`ldd` *file* `/usr/ccs/bin/dump -Lv` *file*	`LD_LIBRARY_PATH`
`sun4`	`ldd` *file*	`LD_LIBRARY_PATH`

D.4.1 Solaris

Under Solaris the `/usr/ccs/bin/dump -Lv` *file* will tell you more shared library information about a *binary*. Under Solaris2, binaries compiled with shared libraries can have a path compiled that is used to search for shared libraries. This path is called the `RPATH`. The `ld` option `-R` is used to set this at compile time. Use `/usr/ccs/bin/dump -Lv` *binary* to view the `RPATH` for *binary*. The `RPATH` for a library can be set at the time of creation with the `-L` flag:

```
g++ -shared -L/users/ptolemy/lib.$PTARCH -o librx.so *.o
```

or by passing the -R flag to the linker:

```
g++ -shared -Wl,-R,/users/ptolemy/lib.$PTARCH -o librx.so *.o
```

Constructors and Destructors between SunOS4.x and Solaris2

The Solaris2 SPARCompiler c++4.0 Answerbook says

> On SunOS 5.x, all static constructors and destructors are called from the .init and .fini sections respectively. All static constructors in a shared library linked to an application will be called before `main()` is executed. This behavior is slightly different from that on SunOS4.x where only the static constructors from library modules used by the application are called.

D.4.2 SunOS

The SunOS4.x port of Ptolemy uses BSD `ld` style linking, which will **not** work with a binary that is linked with **any** shared libraries. For incremental linking of stars to work, the ldd command must return `statically linked` when run on a SunOS4.x `pigiRpc` or `ptcl` binary.

```
ptolemy@mho 2% ldd ~ptolemy/bin.sun4/pigiRpc
/users/ptolemy/bin.sun4/pigiRpc: statically linked
```

D.4.3 HPUX

Under HPUX, shared libraries must be executable or they will not work. Also, for performance reasons, it is best if the shared libraries are not writable.

Under HPUX, shared libraries have a `.sl` suffix, and HPUX uses the `SHLIB_PATH` environment variable to search for libraries.

Under HPUX10, when you are building shared objects, you need to specify both `-fPIC` and `-shared`. (`-fpic -shared` will also work). The reason is that the temporary files that are generated by g++'s collect program need to be compiled with `-fPIC` or `-fpic`. Other platforms don't need both arguments present.

D.5 GateKeeper Error

If there are problems with shared libraries, then you may see

```
ERROR: GateKeeper error!
```

message when you exit `pigi`.

`GateKeepers` are objects that are used to ensure atomic operations within the Ptolemy kernel. The Ptolemy error handling routines use `GateKeepers` to ensure that the error messages are not garbled by two errors trying to write to the screen at once.

Say we have two files that make up two different libraries, and both contain the line:

```
KeptGate gate;
```

With static libraries, the linker will resolve gate to one address and call the constructor once. The destructor will also be called once.

With shared libraries, there are two instances of this variable, so the constructor and the destructor get called twice.

The problem is that there are several different implementations of the error routines depending on if we are running under `pigi`, `ptcl` or `tycho`. The static function `Error::error` is defined in several places, and which definition we get depends on the order of the libraries. (See `$PTOLEMY/src/kernel/Error.[cc,h]`, `$PTOLEMY/src/pigilib/XError.cc`, `$PTOLEMY/src/ptcl/ptclError.cc` and `$PTOLEMY/src/tycho/tysh/TyError.cc`). Each implementation defines a static instance `gate` of `KeptGate`.

You will see the `ERROR: GateKeeper error!` message when you exit if there is more than one `KeptGate gate`, and the destructor is called twice for `gate`.

Under HPUX10.x, the error message is produced if `libptolemy` is static and the other lib (`libpigi`, `libptcl`, `libtysh`) is shared. Here the work around is to make the other library static too.

The way to debug `GateKeeper Error!` problems is to set breakpoints in `Error::error`, and then trigger an error and make sure that the right error routine in the right file is being called. One quick way to trigger an error is to set the current domain to a non-existent domain. Try typing `domain foo` into a `pigi -console`, `ptcl` or `tycho -ptiny` prompt.

Appendix E. Glossary

$PTOLEMY	The directory in which the Ptolemy software is installed.
actor	An atomic (indivisible) function in a dataflow model of computation.
ATM	(1) Asynchronous transfer mode network protocol. (2) A sub-domain of the synchronous dataflow and discrete-event domains to provide the infrastructure for simulating ATM networks.
auto-fork	A fork star that is automatically inserted when a single output is connected to more than one input.
base class	A C++ object that is used to define common interfaces and common code for a set of derived classes. An object may be a base class and a derived class simultaneously.
BDF	A domain using the Boolean-controlled dataflow model of computation. This domain attempts to use compile-time scheduling, but will fall back to run-time scheduling if necessary.
behavioral modeling	System modeling consisting of functional specification plus modeling of the timing of an implementation (cf. functional modeling).
Block	The base class defined in the kernel for stars, galaxies, universes, and targets.
block	A star or a galaxy.
Boolean-controlled dataflow	A model of computation that includes synchronous dataflow, but adds actors that may or may not produce or consume tokens on any given input or output. Whether these actors produce or consume tokens depends on a Boolean signal.
code generation	The synthesis of a standalone implementation in some target language from a network of Ptolemy blocks.
code generation domain	A domain that supports code generation, but not simulation.
CG	A domain that defines many of the base classes and schedulers used in code generation domains. It has no direct application by itself.
CG56	A domain that synthesizes assembly code for the family of Motorola DSP56000 digital signal processors. It uses the synchronous dataflow model of computation.

CG96	A domain that synthesizes assembly code for the family of Motorola DSP96000 digital signal processors. It uses the synchronous dataflow model of computation.
CGC	A domain that synthesizes C code. It uses the synchronous dataflow or Boolean-controlled dataflow model of computation.
CG-DDF	A code generation domain that uses the dynamic dataflow model of computation. This has not been maintained beyond version 0.4.1 of Ptolemy.
codesign	The simultaneous design of the software and hardware composing a system.
communicating processes	A model of computation in which multiple processes execute concurrently and communicate with one another by passing messages.
compile-time scheduling	A scheduling policy in which the order of execution of blocks is precomputed when the execution is started. The execution of the blocks thus involves only sequencing through this precomputed order one or more times (cf. run-time scheduling).
contents facet	An oct facet that defines the physical appearance of a schematic.
CP	A simulation domain using the communicating processes model of computation. Each star forms a process that runs under the Sun lightweight process library.
derived class	A C++ object that is derived from some base class. It inherits all of the members and methods of the base class.
dataflow	A model of computation in which actors process streams of tokens. Each actor has one or more firing rules. Actors that are enabled by a firing rule may fire in any order.
DDF	A simulation domain that uses the dynamic dataflow model of computation.
DE	A simulation domain that uses the discrete-event model of computation. In the DE domain, particles transmitted between blocks represent events that trigger changes in system state. Events carry an associated timestamp, and are processed in chronological order.
discrete event	A model of computation used to model systems that change state abruptly at arbitrary points in time, such as queueing networks, communication networks, and computer architectures. A block is enabled when an event at one of its inputs is the “oldest” event in the system, in that its timestamp has the smallest value. Once enabled, the block may be executed, and in the process may produce more events.
domain	A specific implementation of a model of computation.

Domain	The base class in the Ptolemy kernel from which all domains are derived.
drag	The action of holding a mouse button while moving the mouse.
dynamic dataflow	A model of computation supporting any computable firing rule for actors. This model of computation requires run-time scheduling.
event	A particle generated by a block in a discrete-event model of computation. This particle carries a timestamp.
event horizon	The interface between domains that manages the flow of particles from one domain to another.
facet	A schematic, palette, or icon as represented in `oct`. In `pigi`, a facet is exactly that represented by one `vem` window.
FFT	The Fast Fourier Transform is an efficient way to implement the discrete Fourier transform in digital hardware.
firing	A unit invocation of an actor in a dataflow model of computation.
firing rule	A rule that specifies how many tokens are required on each input of a dataflow actor for that actor to be enabled for firing.
fork star	A star that reads one input particle and replicates it on any number of outputs.
functional modeling	System modeling that specifies input/output behavior without specifying timing (cf. behavioral modeling).
galaxy	A block that contains a network of other blocks.
Galaxy	The class (derived from Block) in the Ptolemy kernel that represents a network of other blocks.
Gantt chart	A graphical display of a parallel schedule of tasks. In Ptolemy, the tasks are the firings of stars and galaxies.
higher-order functions	Functional programming constructs that apply a function a determined number of times to one or more streams of inputs. Examples of higher-order functions from Lisp include `mapcar` and `apply`.
HOF	A domain implementing higher-order functions that are expanded at compile-time and incur no run-time overhead. HOF stars are typically embedded in other domains, and provide graphical expression of parameterized parallel, cascaded, and recursive structures.
homogeneous synchronous dataflow	A particular case of the synchronous dataflow model of computation, where actors produce and consume exactly one token on each input and output.

icon	A graphical object that represents a single block or palette.
interface facet	A facet that defines the physical appearance of an icon (cf. contents facet).
Itcl	[incr Tcl], an object oriented extension to Tcl.
iteration	A set of executions of blocks that constitutes one pass through the precomputed order of a compile-time schedule.
kernel	The set of classes defined in the directory `$PTOLEMY/src/kernel`.
layer	In vem, a color with a given precedence. Colors with higher precedence will obscure colors with lower precedence.
master	In `vem`, the interface and contents facet referred to by an icon. The master is represented by an absolute Unix path name pointing to the directory in which the facet is stored.
masters	A program for examining and changing the list of masters (see above) that make up a schematic or palette.
MDSDF	A simulation domain that uses a multidimensional extension to the synchronous dataflow model of computation. Actors in MDSDF consume data defined on rectangular grids, e.g. a subblock in an image.
member	A C++ object that forms a portion of another object.
method	A function defined to be part of an object in C++.
model of computation	A set of semantic rules defining the behavior of a network of blocks.
net	A graphical connection between ports in vem.
object	A data type in C++ consisting of members and methods. These members and methods may be private, protected, or public. If they are private, they can only be accessed by methods defined in the object. If they are protected, then they can also be accessed by methods in derived classes. If they are public, then they can be accessed by any C++ code.
oct	A design database developed by the CAD Group at U. C. Berkeley. Oct is used to store graphical representations of Ptolemy applications.
octtools	A collection of CAD tools based on the `oct` database. Some programs from the octtools distribution are used within Ptolemy.
palette	A facet that contains a library of icons.
parameter	The initial value of a state.
particle	A datum (e.g. a floating-point value) communicated between blocks.
pepp	The program that translates stars written in the Thor domain to C++.

pigi	The Ptolemy interactive graphical interface. Pigi is implemented as a shell script that starts `vem` and `pigiRpc`.
pigiRpc	The program that forms the core of the Ptolemy graphical user interface. PigiRpc communicates with `vem` via a remote procedure call interface. It is mainly responsible for handling Ptolemy-specific commands from `vem` and translating `oct` representations of Ptolemy systems into a form executable by the Ptolemy kernel.
Plasma	A class in the Ptolemy kernel that serves as a repository for used particles of any particular types. When new particles of the appropriate type are needed, they are taken from the Plasma, if possible, thus avoiding memory allocation.
PN	A simulation domain based on the process networks computational model. Each star forms a process under this domain.
port	An input or output of a star or galaxy.
PortHole	The base class in the Ptolemy kernel for all ports.
ptcl	A textual, interactive command interpreter for Ptolemy. As the name implies, ptcl is based on Tcl.
ptlang	(1) A schema language used to define stars in Ptolemy. (2) The program that translates stars written in the ptlang language to C++.
ptplay	A program to play sound on the workstation speaker.
PTOLEMY	An environment variable with value equal to the name of the directory in which the Ptolemy system is installed.
Ptolemy	A design environment that supports simultaneous mixtures of different models of computation. Ptolemy has been developed at the University of California at Berkeley. The Ptolemy design environment is named after the second century Greek astronomer, mathematician, and geographer.
pxgraph	A plotting program used by several standard Ptolemy stars.
real time	The actual time (cf. simulated time).
RTL	Register-transfer level description of digital systems. This kind of description is used by the Thor domain.
run-time scheduling	A scheduling policy in which the order of execution of the blocks is determined “on-the-fly,” as they are executed (cf. compile-time scheduling).
Scheduler	An object associated with a domain that determines the order of execution of blocks within the domain. Domains may have multiple schedulers.
schematic	A block diagram.
SDF	A simulation domain using the synchronous dataflow model of computation.

Silage	(1) A functional language developed by Paul Hilfinger at U. C. Berkeley for specifying signal processing systems. It is used primarily as input for VLSI synthesis tools. (2) A code generation domain in Ptolemy that synthesizes Silage code and uses the synchronous dataflow model of computation.
simulated time	In a simulation domain, the real number representing time in the simulated system (cf. real time).
simulation	The execution of a system specification (a Ptolemy block diagram) from within the Ptolemy process (i.e., without generating code and spawning a new process to execute that code).
simulation domain	A domain that supports simulation, but not code generation.
snap	In `vem`, an invisible grid defining the points at which graphical objects can have endpoints or corners.
star	An atomic (indivisible) unit of computation in a Ptolemy application. Every Ptolemy simulation ultimately consists of executing the methods of the stars used to define the simulation.
Star	The base class in the Ptolemy kernel for all stars.
state	A member of a block that stores data values from one invocation of the block to the next.
State	The base class in the Ptolemy kernel for all states.
stop time	Within a timed domain, the time at which a simulation halts.
synchronous dataflow	A dataflow model of computation where the firing rules are particularly simple. Every input of every actor requires a fixed, pre-specified number of tokens for the actor to fire. Moreover, when the actor fires, a fixed, pre-specified number of tokens is produced on each output. This model of computation is particularly well-suited to compile-time scheduling.
target	An object that manages the execution of a simulation or code generation process. Thus, for example, in code generation, the target would be responsible for compiling the generated code and spawning the process to execute that code, if desired.
Target	The base class in the kernel for all targets.
Tcl	Tool command language, a textual, interpreted language developed by John Ousterhout at U.C. Berkeley. Tcl is embedded in both `pigi` and `ptcl`.
Thor	A register transfer level digital hardware simulator from Stanford University. Thor is incorporated as a domain within Ptolemy.

timestamp	A real number associated with a particle in timed domains that indicates the point in simulated time at which the particle is valid.
timed domain	A domain that models the evolution of a system in time.
Tk	An X windows toolkit for Tcl. Tk is embedded in `pigi`, which uses it extensively. The interactive sliders, buttons, and plotting capabilities of `pigi` are implemented in Tcl/Tk.
tkoct	An experimental front-end to Ptolemy that replaces `vem`. It is a Tk interface to schematics stored as `oct` facets. It executes universes by calling `ptcl` to evaluate the `ptcl` description of the universe.
token	A unit of data in a dataflow model of computation. Tokens are implemented as particles in Ptolemy.
Tycho	A graphical development environment for Ptolemy that is implemented in Itcl.
universe	An entire Ptolemy application.
URT	The Utah Raster Toolkit for image and video processing. It is used by the image processing stars in the synchronous dataflow domain. The multidimensional synchronous dataflow domain treats images as matrices and does not use the Utah Raster Toolkit.
vem	A graphical editor for objects stored under `oct`. The `vem` schematic capture interface is part of the octtools distribution from U. C. Berkeley, and forms a significant part of `pigi`.
VHDL	The VHSIC hardware description language, a standardized language for specifying hardware designs at multiple levels of abstraction.
VHDLF	A code generation domain for functional modeling of hardware. This domain synthesizes a system description in VHDL.
VHDLB	A code generation domain for behavioral modeling of hardware. This domain synthesizes a system description in VHDL
wormhole	A star in a particular domain that internally contains a galaxy in another domain.

References

[Bha93a] S. S. Bhattacharyya and E. A. Lee, “Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms,” *Formal Methods in System Design*, No. 5, No. 3, December 1994 (Updated from Technical Report UCB/ERL M93/37, EECS Dept., UC Berkeley, May 21, 1993).

[Bha93b] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, “A Scheduling Framework for Minimizing Memory Requirements of Multirate DSP Systems Represented as Dataflow Graphs,” *VLSI Signal Processing VI*, ed. by L. Eggermont, P. Dewilde, E. Deprettre, and J. van Meerbergen, pp. 188-196, IEEE Special Publications, New York, NY, 1993.

[Bha93c] S. S. Bhattacharyya, J. Buck, S. Ha, and E. A. Lee, “Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms,” *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 42, no. 3, pp. 138-150, March 1995 (Updated from Technical Report M93/36, EECS Dept., UC Berkeley, May 21, 1993).

[Bha94a] S. S. Bhattacharyya and E. A. Lee, “Scheduling Synchronous Dataflow Graphs for Efficient Looping,” *J. of VLSI Signal Processing*, vol. 6, December 1993.

[Bha94b] S. S. Bhattacharyya and E. A. Lee, “Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms,” *IEEE Trans. on Signal Processing*, vol. 42, no. 5, May 1994 (Updated from Technical Report UCB/ERL M92/128, EECS Dept., UC Berkeley, November 18, 1992).

[Bha96] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell MA, 1996. (`http://ptolemy.eecs.berkeley.edu/papers/96/softSynthBook/`)

[Bie90] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O’Reilly, G. Sih and E.A. Lee, “Gabriel: A Design Environment for DSP,” *IEEE Micro Magazine*, October 1990, vol. 10, no. 5, pp. 28-45.

[Bla92] Nancy Blachman, *Mathematica: A Practical Approach*, Prentice-Hall, ISBN 0-13-563826-7, 1992.

[Bol91] I. Bolsens, S. De Troch, L. Philips, B. Vanhoof, *et al.*, “Assessment of the Cathedral-II Silicon Compiler For Digital-Signal-Processing Applications,” *ESA Journal*, vol. 15, no. 3-4, pp. 243-260, June 1991.

[Bro88] Randy Brown, “Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem,” *Communications of the ACM*, vol. 31, no. 10, October 1988.

[Buc91] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Multirate Signal Processing in Ptolemy,” *Proc. of Int. Conf. on Acoustics, Speech, and Signal Process-*

ing, Toronto, Canada, April 1991, vol. 2, pp. 1245-1248.

[Buc93a] J. Buck and E. A. Lee, "The Token Flow Model," *Advanced Topics in Dataflow Computing and Multithreading,* ed. L. Bic, G. Gao, and J. Gaudiot, IEEE Computer Society Press, 1993.

[Buc93b] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, Minneapolis, MN, April 1993, vol. I, pp. 429-432.

[Buc93c] J. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph. D. Dissertation, EECS Dept., UC Berkeley, Berkeley CA 94720, September 1993.

[Buc94] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development," January 1994.

[Cha97] W.-T. Chang, S.-H. Ha, and E. A. Lee, "Heterogeneous Simulation -- Mixing Discrete-Event Models with Dataflow," invited paper, RASSP special issue of the Journal on VLSI Signal Processing, January, 1997. (`http://ptolemy.eecs.berkeley.edu/papers/96/heterogeneity`)

[Che94] M. J. Chen, *Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy*, MS Report, ERL Technical Report UCB/ERL No. 94/16, University of California, Berkeley, CA 94720, May 6, 1994.

[Den75] J. B. Dennis, *First Version Data Flow Procedure Language*, Technical Memo MAC TM61, May 1975, MIT Laboratory for Computer Science.

[Edw97] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*, Ph. D. Dissertation, ERL Technical Report UCB/ERL M97/31, EECS Dept., University of California, Berkeley, 1997.(`http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/`

[Eva93] B. L. Evans, *A Knowledge-Based Environment for the Design and Analysis of Multidimensional Multirate Signal Processing Algorithms*, Ph. D. Dissertation, School of Electrical Engineering, Georgia Institute of Technology, Atlanta, GA, June 1993.

[Eva95] B. L. Evans, S. X. Gu, A. Kalavade, and E. A. Lee, "Symbolic Computation in System Simulation and Design," Invited Paper, *Proc. of SPIE Int. Sym. on Advanced Signal Processing Algorithms, Architectures, and Implementations,* July 9-16, 1995, San Diego, CA, pp. 396-407.

[Eva96] G. Arslan, B. L. Evans, F. A. Sakarya, and J. L. Pino, "Performance Evaluation and Real-Time Implementation of Subspace, Adaptive, and DFT Algorithms for Multi-Tone Detection," *Proc. Int. Conf. on Telecommunications,* Istanbul, Turkey, April 15-17, 1996.

[Gen90] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, *et al.*, “DSP Specification Using the Silage Language,” *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, April 1990, vol. 2, pp. 1056-60.

[Ha91] Soonhoi Ha and E.A. Lee, “Compile-Time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration,” *IEEE Trans. on Computers*, vol. 40, no. 11, pp. 1225-1238, November 1991.

[Ha92] S. Ha, *Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs*, Ph. D. Dissertation, EECS Dept., University of California, Berkeley, CA 94720, April 1992.

[Han96] Duane Hanselman and Bruce Littlefield, *Mastering MATLAB*, Prentice Hall, ISBN 0-13-191594-0, 542 pages, 1996.

[Har86] D. Harrison, P. Moore, R. Spickelmier, and A. R. Newton, ‘‘Data Management and Graphics Editing in the Berkeley Design Environment,’’ *Proc. of IEEE Int. Conf. on Computer-Aided Design*, November 1986, pp. 24-27.

[Has93] P. Haskell, *Flexibility in the Interactions Between High-Speed Networks and Communications Applications*, Ph. D. dissertation, EECS Dept., UC Berkeley, Berkeley CA 94720, October 1993.

[Hil85] P. Hilfinger, “A High-Level Language and Silicon Compiler for Digital Signal Processing,” *Proc. of Custom Integrated Circuits Conference*, May 1985, pp. 213-216.

[Hoa78] C. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, August 1978, vol. 21, no. 8, pp. 666-677.

[Hu61] T.C. Hu, “Parallel Sequencing and Assembly Line Problems,” *Operations Research*, vol. 9, no. 6, 1961, p. 841-848.

[Hyl97] C. Hylands, E. A. Lee, and H. J. Reekie, “The Tycho User Interface System,’’ to be presented at the 5th Annual Tcl/Tk Workshop ‘97, Boston, Massachusetts, July, 1997. (`http://ptolemy.eecs.berkeley.edu/papers/97/tcltk-97/`)

[Kah74] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *Info. Proc.*, Stockholm, Sweden, August 1974, pp. 471-475.

[Kah77] G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Info. Proc.*, Toronto, Canada, August 1977, pp. 993-998.

[Kal91] A. Kalavade, *Hardware/Software Codesign Using Ptolemy - A Case Study*, M.S. Report, Electronics Research Laboratory, University of California, Berkeley, CA 94720, December 1991.

[Kal93] A. Kalavade, and E.A. Lee, “A Hardware/Software Codesign Methodology for DSP Applications,” *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 16-28, September 1993.

[Kal94] A. Kalavade and E. A. Lee, “Manifestations of Heterogeneity in Hardware/ Software Codesign”, *Proc. of Design Automation Conference,* San Diego, CA, June 1994.

[Kal96] Asawaree Kalavade, *System Level Codesign of Mixed Hardware-Software Systems,* Tech. Report, UCB/ERL 95/88, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, September 1995.

[Kar66] R. M. Karp and R. E. Miller, “Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing,” *SIAM Journal*, vol. 14, pp. 1390-1411, November 1966.

[Khi94] K. P. Khiar and E. A. Lee, “Modeling Radar Systems using Hierarchical Dataflow,” *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, Detroit, MI, May 8-12, 1995, pp. 3259-3262.

[Lao94] A. Lao, *Heterogeneous Cell-Relay Network Simulation and Performance Analysis with Ptolemy*, M.S. Report, Electronics Research Laboratory, University of California, Berkeley, CA 94720, February 1994.

[Lap91] P. D. Lapsley*, Host Interface and Debugging of Dataflow DSP Systems*, M.S. Thesis, Electronics Research Laboratory, University of California, Berkeley, CA 94720, December 1991.

[Lee87a] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24-35, January 1987.

[Lee87b] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *Proc. of the IEEE.*, vol. 75, no. 9, pp. 1235-1245, September 1987.

[Lee89] E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, “Gabriel: A Design Environment for DSP,” *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, pp. 1751-1762, November 1989.

[Lee91a] E. A. Lee, “Consistency in Dataflow Graphs,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 2, pp. 223-235, April 1991.

[Lee91b] E. A. Lee and J. C. Bier, “Architectures for Statically Scheduled Dataflow,” reprinted in *Parallel Algorithms and Architectures for DSP Applications,* ed. M. A. Bayoumi, Kluwer Academic Publishers, Dordrecht, Netherlands, 1991, pp. 159-90.

[Lee92] E. A. Lee, “A Design Lab for Statistical Signal Processing,” *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, March, 1992, vol. 4, pp. 81-84, San Francisco, CA.

[Lee93a] E. A. Lee, “Multidimensional Streams Rooted in Dataflow,” *Proc. IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium-Grain Parallelism*, January 1993, pp. 295-306, Orlando, FL.

[Lee93b] E. A. Lee, “Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, April 1993, vol. I, pp. 453-456, Minneapolis, MN.

[Lee94] E. A. Lee, "Computing and Signal Processing: An Experimental Multidisciplinary Course," *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. VI, pp. 45-48, Adelaide, Australia, April 1994.

[Lee95] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proc. of the IEEE*, vol. 83, no. 5, pp. 773-801, May 1995 (`http://ptolemy.eecs.berkeley.edu/papers/processNets`).

[Lee96] E. A. Lee and A. Sangiovanni-Vincentelli, "The Tagged Signal Model -- A Preliminary Version of a Denotational Framework for Comparing Models of Computation," ERL Memorandum UCB/ERL M96/33, University of California, Berkeley, CA, 94720, June 4, 1996. (`http://ptolemy.eecs.berkeley.edu/papers/96/denotational/`).

[Mes84a] D. G. Messerschmitt, "A Tool for Structured Functional Simulation," *IEEE J. on Selected Areas in Communications*, vol. 2, no. 1, pp. 137-147, January, 1984.

[Mes84b] D. G. Messerschmitt, "Structured Interconnection of Simulation Programs," *Proc. of Globecom*, November 1984, vol. 2, pp. 808-811, Atlanta, Georgia.

[Mue93] F. Mueller, "A Library Implementation of POSIX Threads Under Unix," *USENIX Conference*, San Diego, CA, January 1993, pp. 29-41.

[Mue95] F. Mueller, "Pthreads Library Interface," Tech. Rep., July, 1995, available by `ftp://ftp.cs.fsu.edu/pub/PART/publications/pthreads_interface.ps.Z`.

[Mur93] P. K. Murthy, *Multiprocessor DSP Code Synthesis in Ptolemy*, Memorandum No. UCB/ERL M93/66, Electronics Research Laboratory, University of California, Berkeley CA 94720, 1993.

[Mur94a] P. K. Murthy, S. Bhattacharyya, and E. A. Lee, "Minimizing Memory Requirements For Chain-Structured Synchronous Dataflow Programs," *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. II, pp. 453-456, Adelaide, Australia, April 1994.

[Mur94b] P. K. Murthy and E. A. Lee, "On the Optimal Blocking Factor for Blocked, Non-Overlapped Schedules," *Proc. of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, November 1994.

[Mur96] P. K. Murthy, "Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow", Ph.D. Dissertation, Tech. Memo UCB/ERL M96/79, Dept. of EECS, Electronics Research Laboratory, Berkeley, Ca 94720, 1996. (`http://ptolemy.eecs.berkeley.edu/papers/96/murthyThesis/`).

[Not91] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: Architecture-

Driven High-Level Synthesis For High-Throughput DSP Applications," *Proc. of the 28th ACM/IEEE Design Automation Conference*, June 1991, pp. 597-602.

[Ous90] J. K. Ousterhout, "Tcl: An Embeddable Command Language," *Proc. of USENIX Conference*, January 1990, pp. 133-146.

[Ous91] J. K. Ousterhout, "An X11 Toolkit Based on the Tcl Language," *Proc. of USENIX Conference*, January 1991, pp. 105-115.

[Ous94] J. K. Ousterhout, *An Introduction to Tcl and Tk,* Addison-Wesley Publishing, Redwood City, CA, 1994, ISBN 0-201-63337-X.

[Par95] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL 95/105, Ph.D. Dissertation, EECS Department, University of California, Berkeley, CA, 94720-1770, December 1995. (http://ptolemy.eecs.berkeley.edu/papers/parksThesis/).

[Pin93] J. Pino, S. Ha, E. Lee, and J. Buck, "Software Synthesis for DSP Using Ptolemy," invited paper in the *J. on VLSI Signal Processing*, vol. 9, no. 1, pp. 7-21, January 1995.

[Pin94] J. L. Pino, T. Parks, and E. A. Lee, "Automatic Code Generation for Heterogeneous Multiprocessors," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, April 1994, vol. II, pp. 445-448, Adelaide, Australia.

[Pin95] J. L. Pino and E. A. Lee, "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, May 1995, pp. 2643-2646, Detroit, MI.

[Pin95] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "*A Hierarchical Multiprocessor Scheduling Framework for Synchronous Dataflow Graphs*," *Proc. of the IEEE Asilomar Conference on Signals, Systems, and Computers,* Pacific Grove, CA, October 29 - November 1, 1995.

[Pin96] J. L. Pino, T. Parks, and E. A. Lee, "Interface Synthesis in Heterogeneous System-Level DSP Design Tools," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, May 1996, Atlanta, GA.

[Rab91] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures," *IEEE Design and Test of Computers*, vol. 8, no. 2, pp. 40-41, June 1991.

[Shi94] S.-I. Shih, *Code Generation for VSP Software Tool in Ptolemy*, MS Report, Plan II, ERL Technical Report UCB/ERL M94/41, University of California, Berkeley, CA 94720, May 25, 1994.

[Sih93a] G. C. Sih and E. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, February 1993.

[Sih93b] G. C. Sih and E. A. Lee, "Declustering: A New Multiprocessor Scheduling

Technique," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 625-637, June 1993.

[Sil91] *Silage User's and Reference Manual*, Prepared by Mentor Graphics/EDC, June, 1991.

[SS92] *Sproc Signal Processor Databook*, Star Semiconductor, 1992

[Sri93] S. Sriram and E. A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, April 1993, vol. I, pp. 345-348, Minneapolis, MN.

[Tho88] R. Alverson, *et al.*, *THOR user's manual: Tutorial and commands*, Technical Report CSL-TR-88-348, Stanford University, January 1988.

[Vai92] P. P. Vaidyanathan, "Multirate Digital Filters, Filter Banks, Polyphase Networks, and Applications: A Tutorial," *Proc. of the IEEE*, vol. 78, no. 1, pp. 56-93, January 1990.

[Wal92] G. Walter, *ATM, Speech Coding, and Cell Recovery*, M.S. Report, EECS Dept., University of California, Berkeley, CA 94720, December 1992.

[Wel96] B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, ISBN 0-13-182007-9, 1995.

[Whi93] K. White, *Xpole: An Interactive, Graphical Signal Analysis and Filter Design Tool*, Tech. Rep. M93/70, Electronics Research Laboratory, University of California, Berkeley, CA 94720, May 1993.

[Wol91] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, ISBN 0-201-51502-4, 1991.

[Won92] Anthony Wong, *A Library of DSP Blocks and Applications for the Motorola DSP96000 Family*, M.S. Report, Plan II, EECS Dept., UC Berkeley, CA 94720, May 1992.

Index