In: IEEE Design and Test of Computers, Sept. 1993, pp 16-28.

Electrical Engineering and Computer Sciences

A Hardware / Software Codesign Methodology for DSP Applications

Asawaree Kalavade
Edward A. Lee

**University of California at Berkeley,
Berkeley, CA 94720.**

Abstract

Embedded systems typically require a mix of hardware and software components. To design these systems, tools should support simultaneous specification, synthesis, and simulation of the software and hardware components. We focus on the design of signal processing systems, where the hardware typically comprises custom datapaths, FSMs, glue logic, and programmable processors, while the software is one or more embedded programs running on the programmable components.

In this paper we describe a systematic heterogeneous design methodology using the Ptolemy framework. Ptolemy is an environment for simulation, prototyping, and software synthesis for heterogeneous systems. The methodology for designing heterogeneous systems with multiple programmable processors and custom hardware is presented with the help of two examples.

1.0 Application-Specific System Design

Application-specific systems are commonly used whenever portability, compactness, cost, or performance requirements cannot be met by general-purpose solutions. Important applications include communications, multimedia systems, consumer products, robotics, and control systems.

Although the term “application-specific” often summons the suffix “integrated circuit”, ASICs are no longer the unrivaled design challenge in research that they once were. Concerted effort in design automation has resulted in sophisticated and widely used tools for routinely designing large special-purpose chips. Although continued improvement of such tools is valuable, much research is refocusing on the system-level design problem.

Two opposing philosophies for system-level design are emerging. One is the *unified* approach, which seeks a consistent semantics for specification of the complete system. The other is a *heterogeneous* approach, which seeks to systematically combine disjoint semantics. Although the intellectual appeal of the unified approach is compelling, we have adopted the heterogeneous approach. We believe that diversity in design styles commonly used today precludes a unified solution in the foreseeable future. Combining hardware and software in a single system implementation is but one manifestation of this diversity.

1.1 Component Subsystems

Even without good design tools, application-specific systems routinely mix diverse design styles. The components commonly found include:

a. **Software or Firmware:** It is rare to find an application-specific system with no software. At the very least, a low-cost microprocessor or microcontroller is used to manage the user interface. But it is common also to implement some of the core functionality in software, often using somewhat specialized programmable processors such as programmable DSPs. Occasionally, an application-specific system is implemented entirely in software. In this case, it is “application-specific” only if reprogramming by the user is not possible.

b. Application-Specific Integrated Circuits: ASIC design has been the focus of synthesis tools, even so-called “high-level” synthesis tools [9], for over a decade. The tools have developed to the point that certain systems can be synthesized fairly quickly. However, this approach to design is not always suitable. Complex, low-speed control functions are often better implemented in software. Moreover, many applications inherently require programmability, for example to customize the user interface. ASICs also cannot accommodate late design changes, and iterations of the design are expensive; thus they may not be suitable for implementation of immature applications. It is increasingly common to use one or more ASICs for the more well-understood and performance-intensive portions of the application, combined with programmable processors to implement the rest.

c. **Domain-Specific Programmable Processors**: Design re-use can drive down development time and system cost. For this reason, introducing enough programmability into a circuit to broaden its base of applications is often advisable. The suitable applications may range from a half dozen different algorithms to an entire domain such as signal processing. The processor itself can be designed by jointly optimizing the architecture, the instruction set, and the programs for the applications [22]. A major drawback to this approach is that it often requires a support infrastructure in the form of software and development systems in order to make re-use viable.

d. Core-Based ASICs: This is an emerging design style, where programmable processor cores are combined with custom datapaths within a single die. Manufacturers of programmable processors are making the cores of their processors available as megacells [6][20] that can be used in such designs. Alternatively, it is possible to use the core of an in-house processor [12]. Such core-based designs offer numerous advantages: performance improvement (due to critical components being implemented in custom datapaths, and faster internal communication between the hardware and software), field and mask programmability (due to the programmable core), and area and power reduction (due to integration of hardware and software within a single core). These designs are especially attractive for portable applications, such as those typically found in digital cellular telephony [21]. The design of such systems requires partitioning the application into hardware and software, and exploring trade-offs in different implementations. Design tools do not currently support this well.

e. **Application-Specific Multiprocessors**: Some intensive applications have high enough complexity and speed requirements to justify development of an application-specific multiprocessor system. In such systems, the interconnect can be customized, along with the software and the selection of processors. Examples of design approaches for such systems range from homogeneous interconnections of off-the-shelf programmable components [5] to heterogenous interconnections of arbitrary custom or commodity processors [32].

Other possible components include analog circuits and field programmable gate arrays. The above components are commonly mixed within a single system design, getting for example, multiple programmable processors along with custom ASICs. Furthermore, the processors need not be of the same kind. The issues in the design of such systems range from partitioning of the algorithm between hardware and software, selection of the type and number of processors, selection of the interconnection network, synthesis of the software (partitioning, scheduling, and code generation), and synthesis of custom hardware.

The design styles involved in the above components are diverse. Tools that synthesize either complete software or complete hardware solutions are common, but tools that support a mixture are rare. In the next subsection we discuss the integration of tools for concurrent hardware/software design.

1.2 What is Codesign?

The simultaneous design of the hardware and software components for such multifarious systems is what we call hardware/software codesign. In a traditional design strategy, the hardware and software partitioning decisions are fixed at an early stage in the development cycle, and the hardware and software designs are developed independently from then onwards. There is little interaction between the two designs due to the lack of a unified representation, simulation, and synthesis framework. The new demands suggest a more flexible design strategy, where the hardware and the software designs proceed in parallel, with feedback and interaction between the two as the design progresses. The final hardware/software split can then be made after the evaluation of alternative structures with respect to performance, programmability, area, power, non-recurring (development) costs, recurring (manufacturing) costs, reliability, maintenance, and evolution of design.

Thus, it is necessary to have tools that support unified hardware/software representation, heterogeneous simulation at different levels of abstraction, and hardware/software synthesis.

1.3 Codesign for DSP Applications

We are developing a codesign methodology applicable to digital signal processing (DSP) and communications systems. DSP applications have the desirable feature that the algorithms are moderately simple, yet they demand high performance and throughput. Furthermore, exploring the trade-offs between different implementations with respect to area, power, cost, and performance is critical for consumer products and portable applications — where DSP is being widely used.

We are focusing on the design of the hardware and software for such systems, where the hardware typically comprises custom datapaths, FSMs, glue logic, and programmable signal processors, while the software is the program running on the programmable components. A variety of commercial DSP microprocessors can be used for most of the sophisticated signal processing required in these applications, while custom hardware can be synthesized for some of the computation-intensive components.

The organization for the rest of the paper is as follows: In Section 2.0 a generic codesign methodology is proposed, followed by a discussion of related research in Section 3.0. The Ptolemy design environment is described in Section 4.0, with a detailed explanation of its support for heterogeneous design styles. In Section 5.0, the codesign methodology under Ptolemy is discussed. Finally, in Section 6.0, two examples are developed in detail.

2.0 A Generic Codesign Methodology

In this section we describe a codesign methodology for designing heterogeneous hardware/software systems. This is a general codesign scheme, and does not apply to any framework in particular. The following discussion refers to Figure 1, and the numbers in parentheses correspond to the corresponding stages in the design methodology.

Given a set of design constraints (real-time requirements, performance, speed, area, code size, memory requirements, power consumption, and programmability), the task is to come up with an optimal design for the software and hardware components that meets the given specifications.

Given a system specification, an algorithm is first developed using high-level **functional simulations** (1) — without any assumptions about implementation (such as available instruction set or register precision). For instance, while designing a modem, a designer would experiment with different algorithms for timing recovery at this stage.

The algorithm is then **partitioned** into hardware and software (2). This partitioning is guided by speed, complexity, and flexibility requirements. Components that need field programmability, or those that are inherently better accomplished in software, are assigned to software implementations. For instance, in the design of a transceiver, a software realization of the coder/decoder would allow changing the constellation easily, enabling the support of multiple modem standards. Similarly, operations whose execution speed is critical are allocated to hardware. Phase detectors for example, can be implemented using the CORDIC algorithm [12], which is suitable for compact VLSI designs. Of course, this partitioning process would have to be iterated in order to explore the design space.

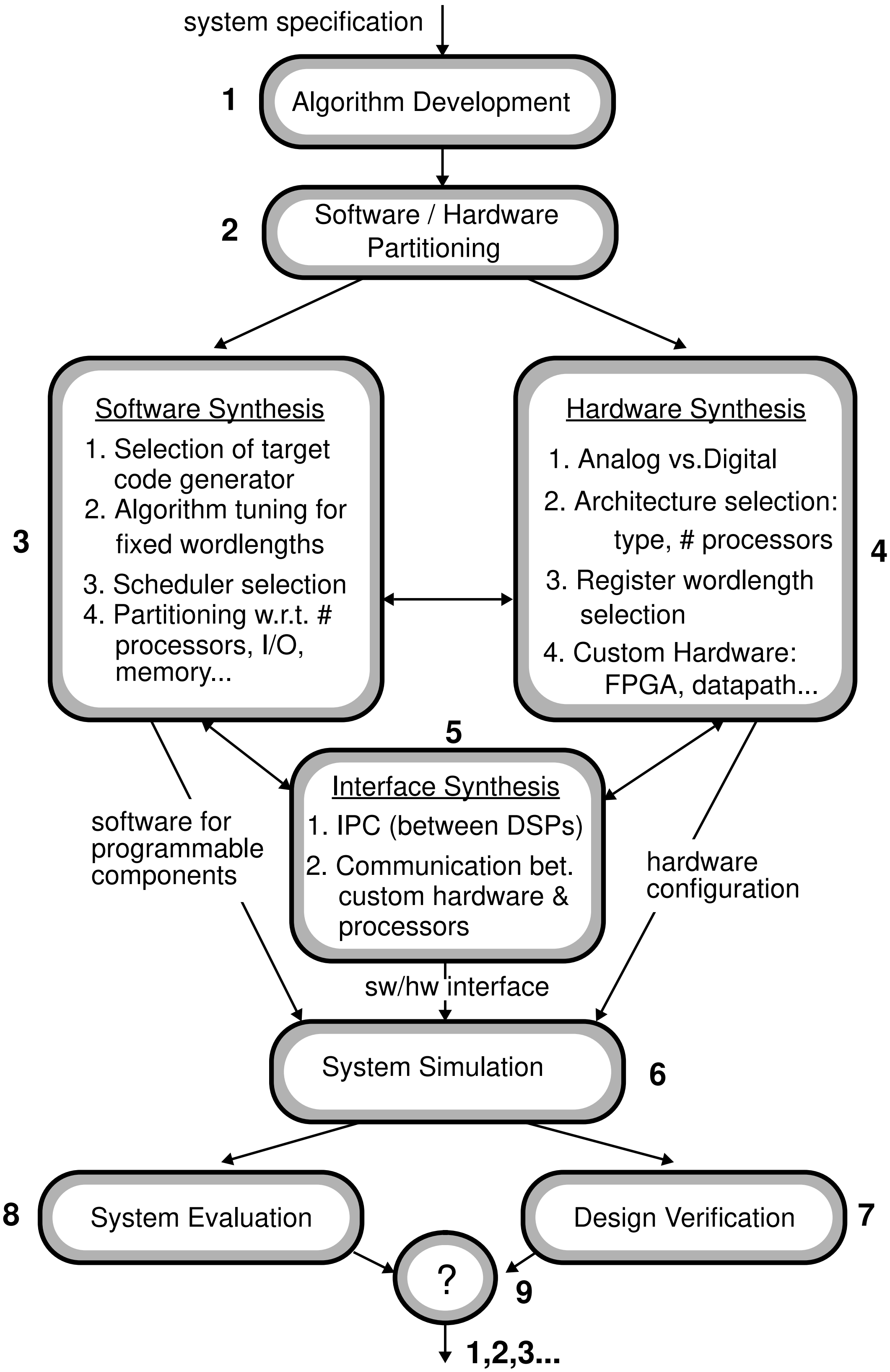

FIGURE 1. A Generic Codesign Methodology

Partitioning is followed by **software** (3), **hardware** (4), and **interface** (5) synthesis. The three are closely coupled — changing one has immediate effects on the other. Hardware design involves a number of design decisions: selection of the programmable processor (this directly affects selection of the code generator), determination of the number of processors and their connectivity (this influences partitioning of the code and the synthesis of the software/hardware interfaces). When dealing with custom hardware synthesis, the choices range from generating custom datapaths to generating masks for FPGAs. While designing custom datapaths, it is necessary to choose the register wordlengths. By selecting some hardware structures (filter realizations, for instance), performance requirements may be met with register widths smaller than those estimated for other structures.

On the software front, in the case of fixed point processors, some algorithmic modifications might be necessary to minimize the effects of finite precision (such as limit cycles and quantization errors). Software synthesis involves partitioning and scheduling the code on multiple processors, and synthesizing the code for communicating between different processors — these decisions depend on the architecture selected. Partitioning among different processors is accomplished by optimizing different cost functions such as: interprocessor communication cost, memory bandwidth, and local and global memory sizes. The interface synthesis involves adding latches, FIFOs, or address decoders in hardware, and inserting code for I/O operations and semaphore synchronization in software. Thus the typical way to go about solving this cyclic problem is to start with a design and iteratively work on it by exploring different options.

Once the hardware and software components are synthesized, the next step is a **heterogeneous simulation** (6). In particular, the simulated hardware must run the generated software. This involves an interaction between a number of different simulators if different specification languages are used.

Results of the simulation are then used for verifying (7) that the design meets the specifications. Having performed the hardware and software synthesis for a particular design choice, it is possible to come up with estimates of the area, power, critical path, component and bus utilization, etc. These estimates are used to evaluate the design (8) and re-partition the system to try out different options (9).

Thus, the entire process is iterative and the primary need is for a unified framework that allows the hardware and software components to be integrated from the specification through the synthesis, simulation, and evaluation phases. In Section 4.0. we describe the Ptolemy design environment, which aims towards such a unified framework.

3.0 Related Work

Having discussed a generic codesign methodology, in this section we attempt to briefly summarize the current status of research in this area.

In [12], an approach to the design of signal processing systems on heterogeneous IC architectures is presented. The CATHEDRAL system is fairly mature in the areas of accelerator datapath synthesis and retargetable generation of microcode for DSP cores; however, the problems of hardware/software partitioning and interface synthesis have not yet been addressed in much detail in their work. Gupta et al. [13] address the partitioning problem for embedded controllers within the Olympus synthesis system. Starting with a HardwareC description of the system, they generate a constraint-driven graph, and partition it into software (running on a programmable processor) and hardware (multiple ASICs). Points of non-determinism (caused by either external synchronization or internal data-dependent delay operations) are identified, and by decoupling the internal and external points of non-determinism, the system is partitioned across hardware and software. Chou et al. [10] address the problem of synthesizing the hardware/software interface for microcontroller-based applications. In [22], Paulin et al. describe codesign for application specific programmable processors within the SADES environment. The designer begins with the design and selection of an instruction set. Code for this target instruction set is synthesized, and the application is run on a VHDL model of the synthesized processor. In [24] Prakash et al. describe a scheduling driven approach for the synthesis of application-specific multiprocessor systems. The application is partitioned onto heterogeneous processors that differ either in style or speed. Timing constraints are used to formulate a mixed integer-linear programming problem, which is solved to generate a static task schedule, a description of the architecture, and a mapping of the tasks onto the processors. In [30], a system design methodology for real-time signal processing is described within the HYPERSPACE environment, which integrates a library of architecture specific compilers, partitioning, estimation, and performance analysis tools. Yet another design framework (SIERRA) for application-specific systems is described in [28]. A high-level description of the system as a network of processes is mapped to a system architecture template consisting of multiple boards using dedicated hardware modules and ASICs as well as software processes running on programmable hardware modules.

In this work we address the problem of heterogeneous system specification, simulation, and synthesis, within the Ptolemy environment.

4.0 The Ptolemy Framework

Ptolemy [1][8][23][15][7] is an environment for simulation and prototyping of heterogeneous systems. It uses object-oriented software technology to model each subsystem in its most natural and efficient manner, and has mechanisms to integrate heterogeneous subsystems into a whole. The infrastructure of the Ptolemy kernel is presented first, followed by a discussion of the multi-paradigm simulation and retargetable code generation capabilities of Ptolemy.

4.1 Internal Structure of Ptolemy

Figure 2 shows the structural components of Ptolemy. The basic unit of modularity in Ptolemy is the Block. Portholes provide the standard interface through which Blocks communicate. A Block contains a module of code (the “go()” method) that is invoked at run-time, typically examining data present at its input Portholes and generating data at its output Portholes. The invocation of “go” methods is directed by a Scheduler that determines the operational semantics of a network of Blocks. Blocks communicate using streams of Particles, which form the base type for all messages passed. The Geodesic class establishes the connection between Portholes. The Plasma class manages the reclamation of the used Particles.

The lowest level (atomic) objects in Ptolemy are of the type Star, derived from Block. A Galaxy, also derived from Block, contains other Blocks internally. A Galaxy may internally contain both Galaxies and Stars. A Target, also derived from Block, controls the execution of an application. In a simulation-oriented application, it will typically invoke a Scheduler to manage the order in

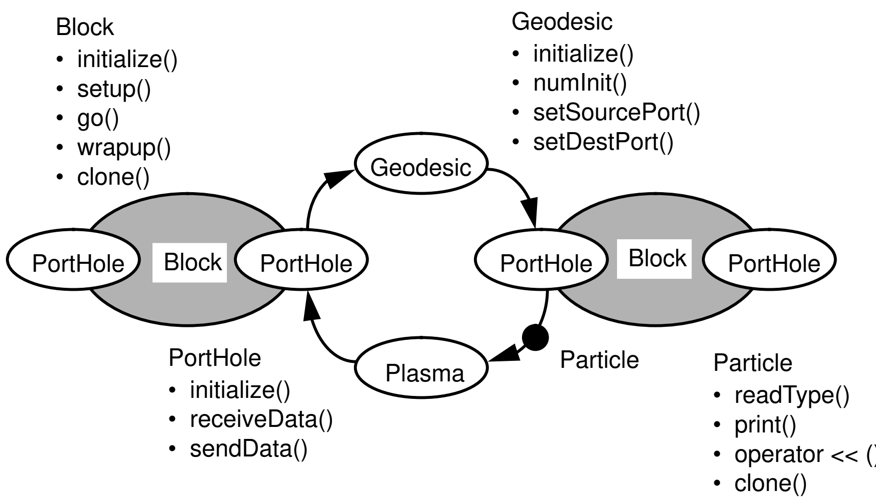

FIGURE 2. Block objects in Ptolemy send and receive data encapsulated in Particles to the outside world through Portholes. Buffering and transport is handled by the Geodesic and garbage collection by the Plasma.

which Star methods are invoked. For a synthesis-oriented application it can synthesize assembly code for a programmable DSP, invoke the assembler, and run a simulation. A Universe, which contains a complete Ptolemy application, is a type of Galaxy.

4.2 Heterogeneous Simulation using Ptolemy

Ptolemy accomplishes the goal of multiparadigm simulation by supporting a plethora of different design styles encapsulated in objects called Domains. A Domain realizes a computational model appropriate for a particular type of subsystem. A Domain in Ptolemy consists of a set of Blocks, Targets, and associated Schedulers that conform to a common computational model — the operational semantics that govern how Blocks interact with one another. The Domain and the mechanism of coexistence of Domains are the primary features that distinguish Ptolemy from otherwise comparable systems such as Comdisco's SPW and Bones, Mentor Graphic's DSPstation, and the University of New Mexico Khoros System [26]. Some of the simulation domains that are currently supported include 'Synchronous Data Flow' (SDF) [18], 'Dynamic Dataflow' (DDF), 'Discrete Event' (DE), and the 'Digital Hardware Modeling Environment' (Thor) [29].

The Domain class by itself gives Ptolemy the ability to model *subsystems* differently, using a model appropriate for each subsystem. It also supports mixing these models at the system level to develop a heterogeneous system with different levels of abstraction. The mixture is hierarchical. Any model of computation can be used at the top level of the hierarchy. Within each level of the hierarchy, it is possible to have blocks containing foreign domains. The object-oriented principles of polymorphism and information hiding are used to get interoperability in a modular way. This hierarchical heterogeneity is quite different from the concept of a *simulation backplane*, implemented for example in Viewlogic's SimBus system. A simulation backplane imposes a top-level model of computation through which all subsystems interact. Figure 3 shows the top view of a Universe associated with a certain Domain called "XXX", associated with which are XXXStars and a XXXScheduler. A foreign subsystem that belongs to Domain YYY, and has its own set of YYYStars and a YYYScheduler is embedded in this XXXDomain Universe. This foreign subsystem is contained entirely within an object called an XXXWormhole. An XXXWormhole is a type of XXXStar — so at its external interface it obeys the operational semantics of the external domain, but internally it consists of an entire foreign subsystem. A Wormhole can be introduced into the XXX domain without any need for the XXXScheduler to know of the existence of the YYY domain. The key to this interoperability is the interface between the internal structure of a Wormhole and its external environment. This interface is called the EventHorizon. The EventHorizon is a minimal interface that just supports exchange of data and permits rudimentary standardized interaction between schedulers. Each domain provides an interface to the EventHorizon, and thus gains an interface to any other domain. More details on the operation of EventHorizons can be found in [1] and [8]. A complete description of the Ptolemy kernel can be found in [7].

A second mechanism in Ptolemy supports heterogeneity in a different way. A synthesis or simulation operation is managed by an object called a Target. For example, if we are synthesizing a software implementation, then the Target is responsible for generating the code, compiling or assembling it, possibly downloading it into attached hardware for execution, and managing the execution. Instead of attached hardware, however, the Target can consist of simulated hardware implemented in another Ptolemy Domain. The Target, therefore, is the executive manager of a cosimulation.

The SDF and Thor simulation domains that are used in the methodology discussed in this paper are now described in further detail.

Synchronous Data Flow (SDF) Domain: SDF [18] is a data-driven, statically scheduled domain. “Data-driven” means that the availability of Particles on the inputs of a star enables it. Stars with no inputs (“sources”) are always enabled. “Statically scheduled” implies that the firing order of the stars is determined only once during the start-up phase and this schedule is periodic. The SDF domain supports simulation of algorithms, and also allows functional modeling of components such as filters and signal generators [16].

Thor Domain: The Thor Domain implements the Thor simulator, which is a functional simulator for digital hardware [29]. It supports the simulation of circuits from the gate level to the behavioral level. Thor thus provides Ptolemy with the ability to simulate digital components ranging in

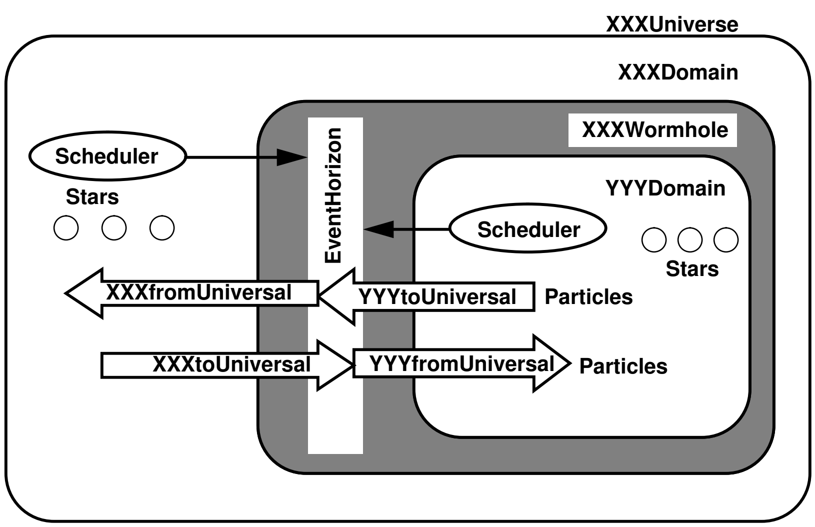

FIGURE 3. The universal EventHorizon provides an interface between the external and internal domains.

complexity from simple logic gates to programmable DSP chips [3]. It is essentially equivalent to many event-driven register-transfer level circuit simulators available commercially. It is used in preference to a commercial system only because, in our experimental framework, we require access to the source code, and we wish to freely distribute our software to disseminate the ideas embodied in it.

A Thor model for the Motorola DSP56000 has been developed under Ptolemy. The "setup()" method of the DSP Star under Thor simply establishes a socket connection with *Sim56000*, Motorola's stand-alone simulator for the DSP56000. *Sim56000* is different from most other processor simulators. It is a complete "behavioral simulator" for the processor, not just an "instruction set simulator" or a "bus functional model". This means that it accurately models the behavior of each of the processor's signal pins, while executing the code. Instruction set simulators do not support this feature of modeling the pin-level behavior. Bus functional models just emulate a given pattern of the bus activity, they do not execute any code.

During its "go()" method, this Star translates the logic values present at the processor's pins into values meaningful to the simulator, transfers them to *Sim56000*, and commands the simulator to advance the simulation by one step. It waits for the simulator to transmit the new logic values back to the processor pins and continues with the rest of the simulation. The simulation can be halted at any time by interrupting the simulator window and intermediate register contents can be examined.

Figure 4 illustrates this behavior. The figure represents the simulation of a single DSP system within Ptolemy. The hardware design consisting of a single processor is developed using the Thor and SDF domains of Ptolemy (lower left). The *Sim56000* that is invoked when the system is run can be seen on the lower right. The window in the top left corner shows a Thor logic analyzer monitoring the output on the serial port of the DSP. The Ptolemy code for the DSP block is shown on the top right corner.

Besides processors and digital logic, it is also necessary to model analog components such as A/D and D/A converters, and filters that operate in conjunction with this digital hardware. These analog components can most conveniently be represented by their "functional models" using the SDF domain. Often abstract functional modeling of components such as filters is sufficient — detailed behavioral modeling is not needed — particularly if an off-the-shelf component with well-understood behavior will be used in the final implementation. The Wormhole mechanism discussed above is used to mix the data-driven, statically-scheduled SDF models of analog components with

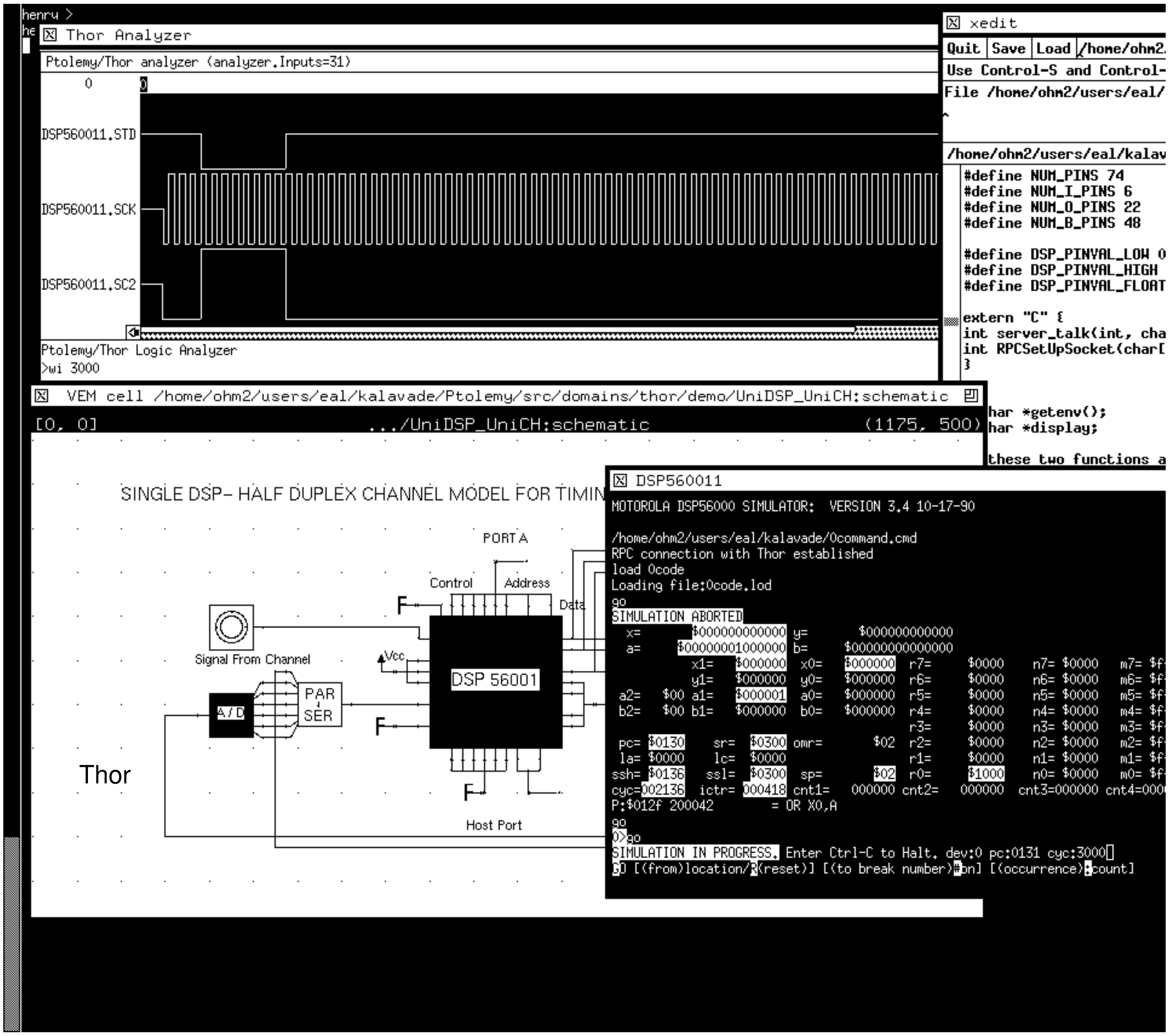

FIGURE 4. An illustration of the Ptolemy simulation environment. A hardware design (bottom left) containing a programmable DSP can be developed using the Thor and SDF Domains. When the DSP Star (its C++ code can be seen in upper right window) is run, it invokes the Motorola DSP56000 Simulator (bottom right) that executes the code. Timing verification is possible using the Thor Logic Analyzer (top left).

event-driven, logic-valued Thor models of digital components within a single simulation. Thus, analog and digital hardware modeling at different levels of abstraction is possible using Ptolemy.

4.3 Retargetable Code Generation under Ptolemy

Given a block diagram description of an algorithm, “code generation” refers to the synthesis of software corresponding to the given algorithm. This software could be synthesized for a variety of target processors. We have implemented assembly code generators for the Motorola DSP 56001 (a 24 bit fixed point processor) and the DSP 96002 (a floating point processor), and have also implemented C and C++ code generators that are suitable for a wider variety of processors. The target processors can take a range of configurations: uniprocessor systems, shared memory parallel architectures, shared bus parallel architectures, and message passing architectures. The Target object encapsulates any code generation capabilities that are peculiar to a particular target architecture, such as the interprocessor communication mechanism. Thus it is possible to synthesize software for the new configuration just by specifying a new Target.

In Ptolemy, code generation domains are subdivided according to the language synthesized. Hence, an SDF domain synthesizing C code is called CGC (code generation in C), while a domain synthesizing assembly code for the Motorola DSP56000 family is called CG56. Each such domain has a simple default Target that defines how the code generated for individual stars is collected, specifies and allocates resources such as memory, defines code necessary for initialization of the platform, and optionally compiles or assembles it. More elaborate Targets can be derived from this. These more elaborate targets generate and run code on specific hardware platforms or on simulated hardware. Figure 5 shows a derivation tree for such targets.

4.3.1 Code Generation for Programmable Processors (Software Synthesis)

The CG56 and CG96 domains are used for the synthesis of assembly code for architectures (specified by the Target) consisting of one or more programmable DSPs (the Motorola 56000 and 96000 respectively). Alternatively, the C code generation domain may be used to synthesize C code, which could be compiled to the desired target processor. Multiprocessor code generation has also been implemented. A suite of schedulers have been implemented in Ptolemy that use properties of the SDF model of computation to partition the code onto multiple processors, schedule the execution of the code, and insert code for interprocessor communication. For example,

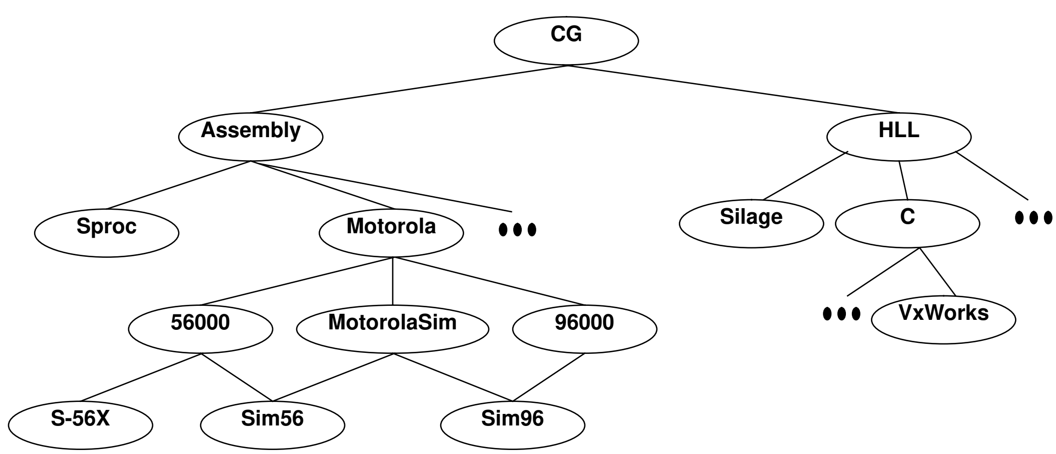

Figure 5. Inheritance Tree for Single Processor Targets.

MultiSim56000 is a multiprocessor target that controls code generation for a multiprocessor system in a shared memory configuration (Figure 8). The number of processors and the address for the shared memory are provided to this target. The target then invokes the appropriate parallel scheduler that partitions and schedules the code onto the processors, inserts semaphore synchronization code, and generates assembly code for each of the processors. An example in Section 6.1 discusses this target in further detail. The design of heterogeneous multiprocessor targets, in which more than one type of programmable processor can be used, is also underway. Parallel schedulers that use different cost functions for partitioning and scheduling the code on heterogeneous programmable components have been developed [27].

4.3.2 Code Generation for Hardware Synthesis

A Silage[14] code generation domain has also been developed under Ptolemy. Silage is a functional or applicative language (each operation can be thought of as applying a function on a set of inputs and generating a set of outputs). It is possible to specify the numerical precision for the inputs and outputs of these functions, as well as the precision for internal computations. In addition, multirate computations such as downsampling and upsampling can be specified is Silage. This makes Silage an attractive language for high-level specification of DSP applications. Furthermore, a number of high-level synthesis systems that use Silage for specification of their inputs are available [25] [12]. As a result, the Silage code generated by Ptolemy provides a link to these synthesis tools, thereby permitting custom hardware synthesis. Thus, the function of the Silage domain is two-fold — synthesis of custom hardware, and bit-true modelling of custom-synthesized hardware.

When a Silage galaxy is nested in a SDF Universe, the blocks in the SDF domain send data to the Silage galaxy. On processing this data, the Silage galaxy generates outputs that can be further processed in the SDF domain. Thus Silage galaxies in the SDF domain represent function application. When such a SDF-Silage system is run:

1. In the setup() phase, Silage code is generated. This code is then fed to high-level synthesis tools such as Hyper [25] to synthesize a custom datapath. Estimates of the critical path, power consumption, and area are obtained. The Silage galaxy is then automatically replaced by a single Silage star (generated by compiling the Silage code, and dynamically linking it into the running simulation). The portholes of this Silage star are of type “fix”. The precision of the data along these portholes, as well as the precision of intermediate results can be specified. This is used to run bit-true simulations to experiment with different wordlengths.

2. In the go() phase, the complete system is simulated. The new Silage star (corresponding to the Silage galaxy) models the functionality of the physical hardware. If Silage is nested within a timed domain (such as Thor), the critical path estimates obtained from the synthesis tools are used to model the hardware execution delay.

The Silage domain thus permits: a) high-level simulation (for functional verification), b) bit-true simulation (for analysis of finite precision effects, fine-tuning of the algorithm for finite wordlengths, and determination of optimal wordlengths), and c) synthesis of custom datapaths for parts of the algorithm committed to hardware implementation. Figure 6 illustrates these ideas. An SDF Universe is shown at the top of the figure. This contains a Silage galaxy that implements a fifth order filter. The filter is realized (in the Silage domain) as a cascade of two biquad sections followed by a first order section. The discrete implementation of the biquad is also shown in the figure. The Silage code generated by Ptolemy is shown on the right, and the layout and hardware estimates for the custom datapath for the biquad section (generated by Hyper) are shown on the lower left and bottom right respectively.

5.0 Ptolemy and Hardware/Software Codesign

In Section 2.0, a generic codesign methodology for designing systems containing a mixture of hardware and software components was proposed. In this section we briefly show how the different components of Ptolemy (Section 4.0) can be applied to hardware/software codesign as defined by the generic methodology.

Figure 1 from Section 2.0 is redrawn in Figure 7. The different domains of Ptolemy that are used for the different phases in the codesign process are shown.

The SDF domain is used for **high-level simulations** and for the development of algorithms (1). The **partitioning** between hardware and software (2) is done manually, with the designer specifying whether a block is to be implemented in hardware or software. Using this information, wormholes are generated for each of the two types of implementations — the components of the algorithm to be implemented in custom hardware are grouped together in a Silage wormhole, while the parts of the algorithm to be implemented as software running on programmable processors are clustered into a CG wormhole corresponding to the target processor (CG56, CG96, etc.,).

A preliminary **hardware design** decision (3) is then made regarding the number of processors to be used (further simulations may prove that this was not a good choice, and changes can be made iteratively). In the case of multiprocessor systems, the system configuration is then selected. This

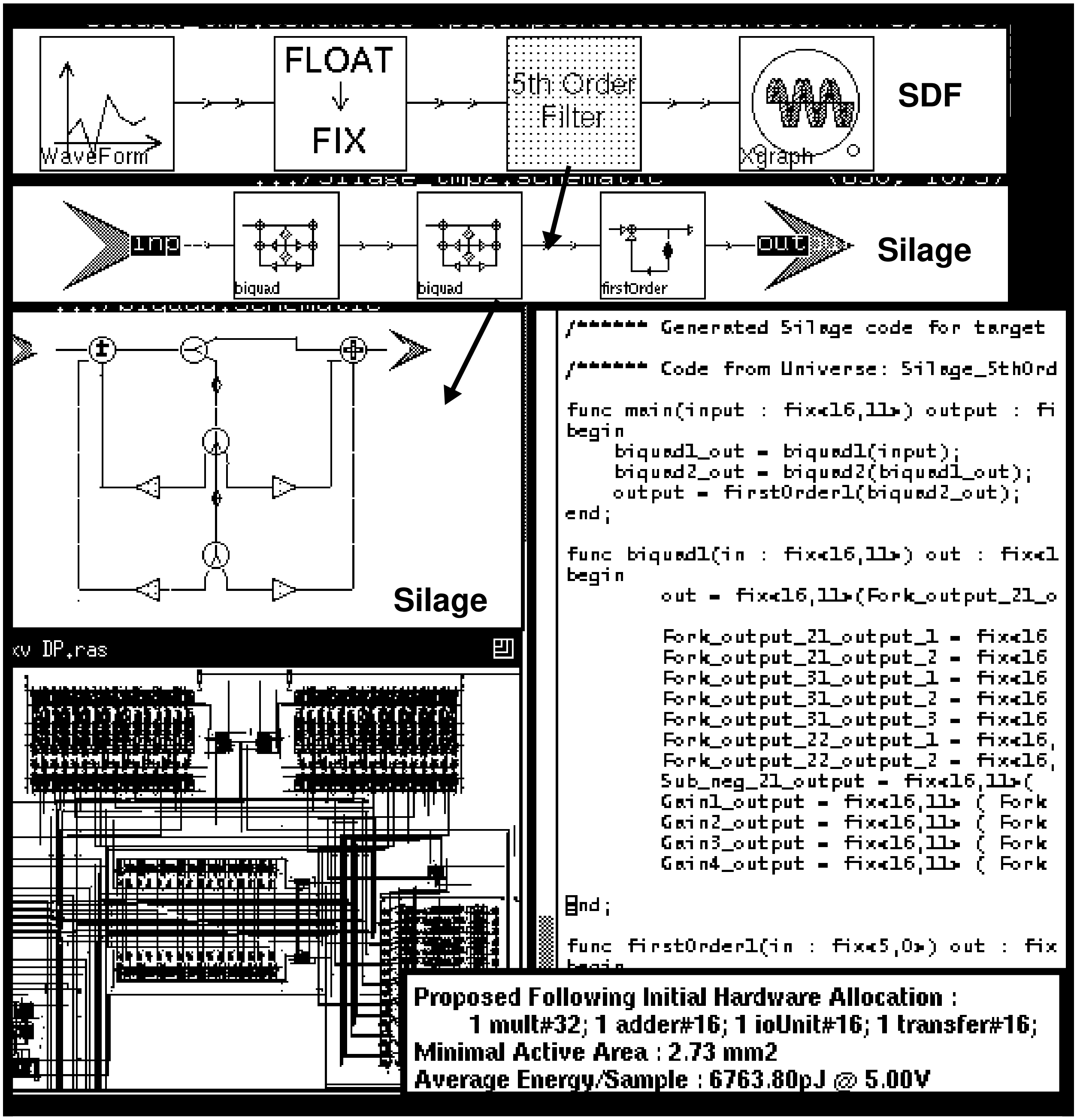

FIGURE 6. A Fifth Order Filter in Silage: The top window shows the SDF Universe with a Silage galaxy for a 5th order filter. The 5th order filter is implemented in the Silage domain as a cascade of two biquad sections and a first order filter. The biquad itself is implemented using discrete components. The generated Silage code, layout, and hardware estimates for the system can be seen on the right, bottom left, and bottom right respectively.

determines the interface (5) between the DSPs (such as a shared multi-ported memory, a shared bus, or serial communication). A Thor simulation model for this architecture is next constructed — this uses Thor models for the DSP chips and glue logic. Functional models for analog components such as A/D convertors and analog front-ends are developed in SDF and added to this Thor model as wormholes. Bit true simulations for the components allocated to custom hardware

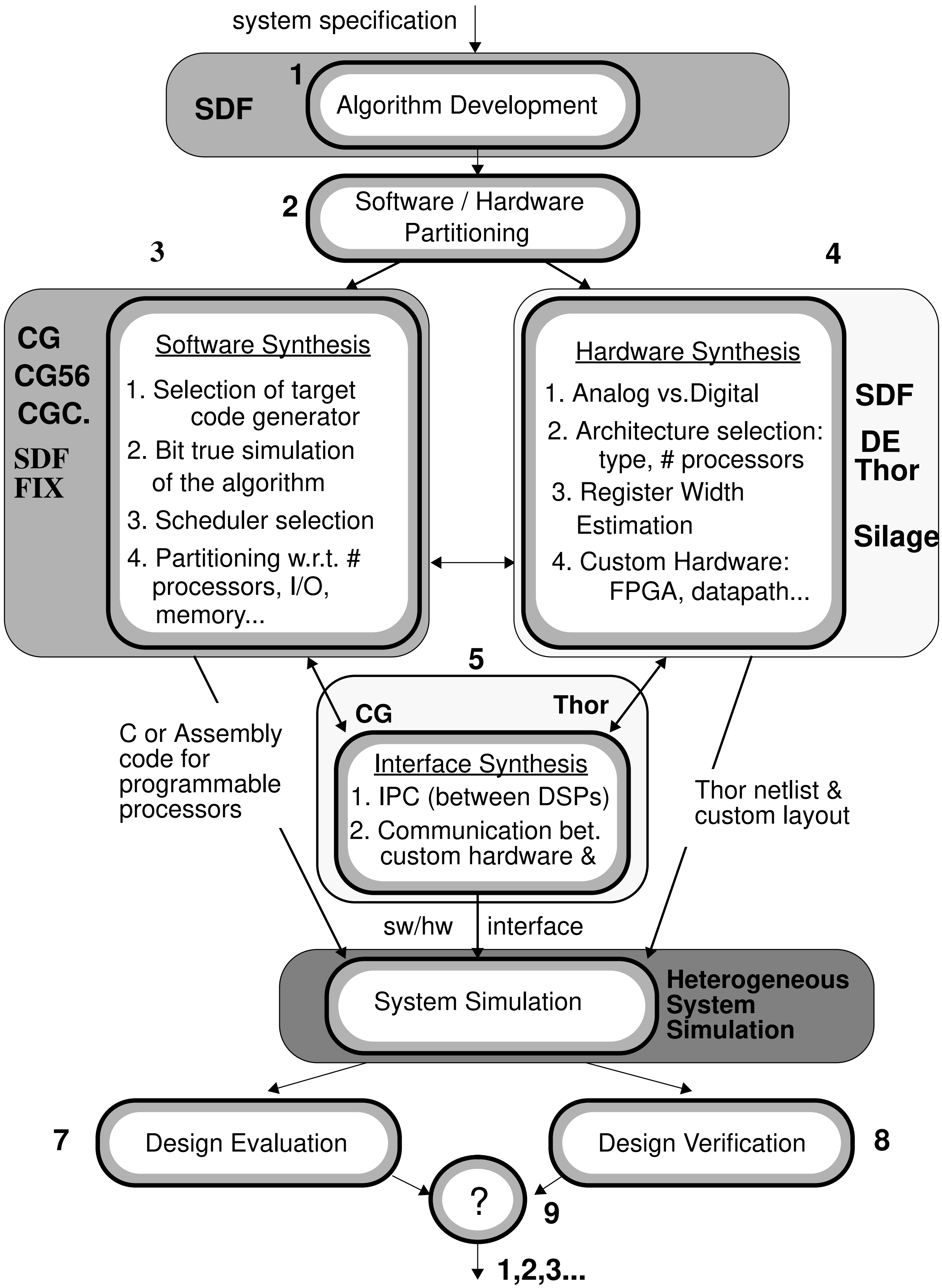

FIGURE 7. The Codesign Methodology under Ptolemy

implementation are carried out in the Silage domain. These simulations give an estimate of the optimal word lengths. Finally, the generated Silage code is fed to synthesis tools to estimate the critical path, power, and area. The Silage blocks are added to the system simulation model to represent the custom hardware.

On the software (4) end, selection of the programmable processor decides the domain for **code generation**. The interface (5) between the hardware and software components is then selected. For instance, the DSP can communicate with external hardware either over one of its serial ports or through memory-mapped I/O. Based on this hardware/software interface and the inter-processor interface, an appropriate code generation target is selected and assembly code is generated. Interface and synchronization code are then inserted in the program. At the same time, hardware components for the interface are added. For example, in the case of serial I/O, a serial-to-parallel register and appropriate clocking circuitry is needed. Similarly, for memory-mapped i/o, address decoders and latches are inserted.

The code size is then analyzed to determine whether or not the program fits in the on-chip program memory. If not, then an external memory is required and a corresponding Thor model for the memory is added to the hardware model. Alternatively, different schedulers may be tried (to replace multiple instances of the code by efficient loops [2], for example), or partitioning may be redone to reduce the software load (by shifting some components to hardware). The number of cycles required to execute the critical sections of the code is determined in order to identify the computationally intensive components. The time available between consecutive samples is compared to the number of cycles required to process one sample to make sure real-time requirements are met. If not, then the system may either be re-partitioned or alternative configurations may be selected.

Next, a **mixed domain simulation** using the various components synthesized so far is run. The processors and glue logic are simulated in the Thor domain, custom-synthesized hardware is modeled via a bit-true Silage galaxy (with execution delay equal to the critical path), and analog components are represented by their functional models in SDF. The DSPs run the programs synthesized by the code generator.

Results of the simulation are manually examined to check whether the performance constraints are met. Other design choices can be explored by repeating the procedure described above. The final selection can be made on the basis of the requirements of the user.

At this point, the infrastructure for these phases of the design process is available within Ptolemy. Work is underway towards automating some of these stages. We are developing a *Design Assis-*

tant that will automatically create the wormholes for the hardware and the software blocks, and insert the interfaces. The Design Assistant would also assist in the analysis and verification (which are currently done manually) and enable exploration of the design space easily. It is important to note that the Design Assistant would “formalize” and “partially automate” the design process, building upon the basic heterogeneous simulation and retargetable code generation facilities that already exist with Ptolemy.

6.0 Examples of Hardware/Software Codesign using Ptolemy

In this section we demonstrate the codesign methodology under Ptolemy with the help of specific examples. The first example illustrates a methodology for multiprocessor system design, while the second one targets a programmable DSP core-based ASIC design.

6.1 Multiprocessor System Design Methodology

Multiple processors are required in many signal processing applications in order to meet real-time constraints. We have selected a demonstration application that barely requires more than one processor for a fully functional implementation. Of course, as the technology improves, one processor will eventually be more than adequate for this particular application. Nonetheless, it has just enough complexity to illustrate our methodology.

The application and its specification are described in Section 6.1.1, followed by a discussion (in Section 6.1.2) of the design methodology followed for the design of this system.

6.1.1 Application Description and Algorithm Specification

Consider the design of a full-duplex telephone channel simulator [15][16][17]. It models the behavior of a physical telephone channel by introducing impairments (as characterized by the EIA-496-A standard [11]) such as linear distortion, frequency offset, phase jitter, non-linear distortion, and noise. It is used by voiceband data modem designers to test the performance of modems under a variety of channel responses. Satisfactory performance of modems under these impairment conditions provides robust assurance of modem performance on most telephone lines in the public-switched telephone network.

The goal is to design an implementation for such a bidirectional channel simulator. The algorithm for one direction of the full-duplex channel simulator (in SDF) is shown in the top window (1) in Figure 8. Similar processing would be performed on the signal coming in the opposite direction.

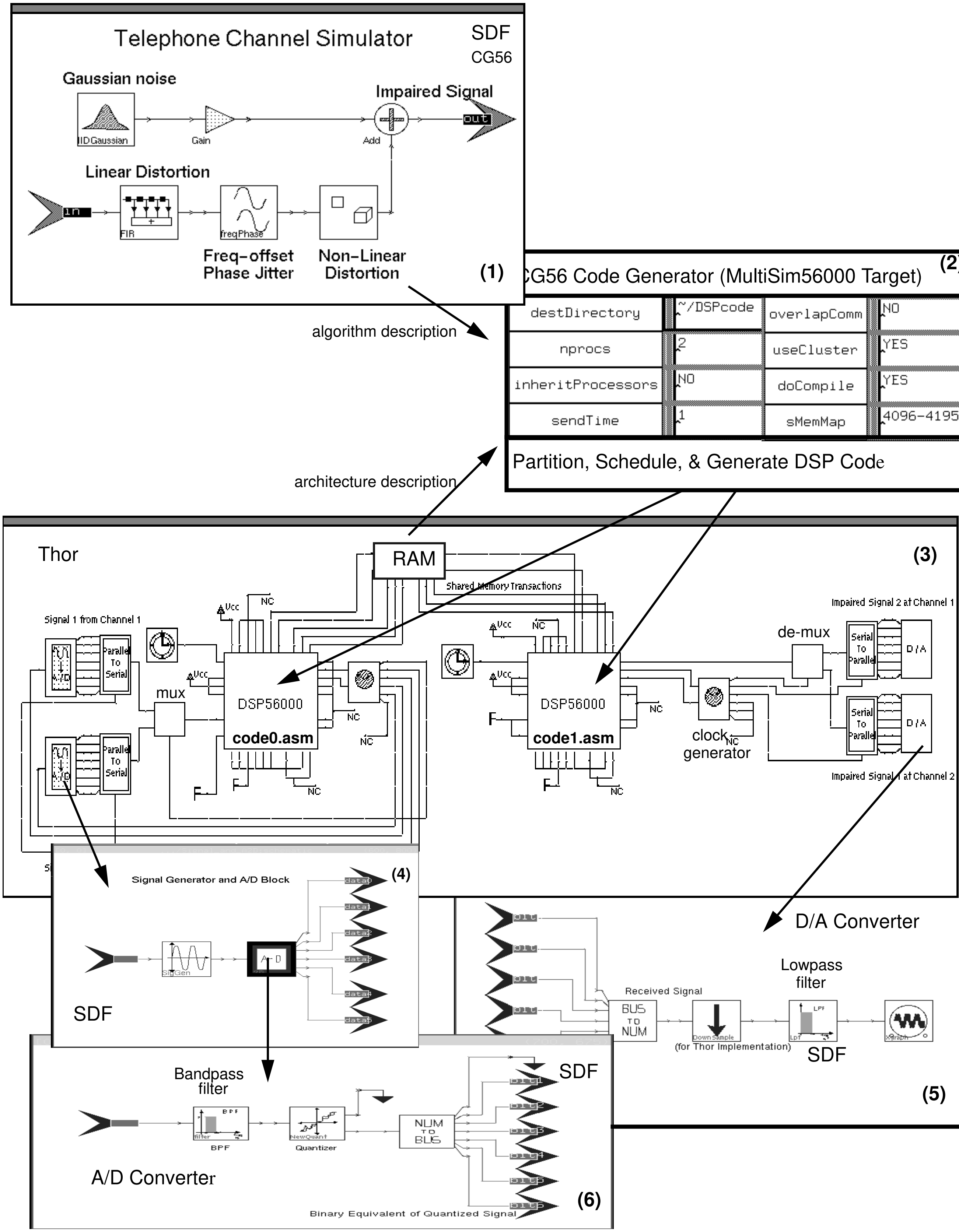

FIGURE 8. Telephone Channel Simulator: The algorithm specification (SDF domain), code generator (CG56 domain, MultiSim56000 Target), and the digital (Thor) and analog (functional models in SDF) hardware components of the design.

6.1.2 System Design

In order to test the modems under different channel conditions, it is necessary to be able to change the degree of various impairments. As a result, it is desirable to incorporate as much functionality in software as possible. As a first cut, the algorithm is partitioned so that it is entirely implemented in software. The Motorola DSP 56001 is selected as the target processor. As a conservative choice, the full-duplex (bi-directional) telephone channel simulator is designed around two DSPs, each handling one direction of impairments. The hardware thus consists of two DSPs, A/D- D/A convertors, and other glue logic, while the software is the program running on the two processors.

A simplistic design is to have the two processors run identical code, each impairing the signal in one direction. The algorithm is provided to a single processor (Motorola 56001) code generator. The code generator synthesizes code for one processor, and the same code is run on the other processor. However, on analyzing the generated code it is found that the code size is more than the capacity of on-chip program memory of the DSP. As a result, two alternatives are possible: use an external program memory for the two processors, or partition the code onto the processors so that each implements a part of the algorithm (thereby requiring less code space). The second option seems more cost effective and is explored next. In this case, the first processor creates some distortion in the signal received from the sending modem, and passes the partially impaired signal to the other processor. The second processor then adds the remaining distortion components and sends the impaired signal to the receiving modem.

The next design decision is the selection of the interface between the two processors. Again, different options are available: communication over the serial port, or use of a shared memory. The latter is selected, and a hardware model for the system is built. The algorithm is presented to the code generator once again, with the new target (MultiSim56000): a two-processor system, with the processors communicating via a dual-ported shared memory. Figure 8 illustrates this design flow. The top window (1) shows the algorithm description. This is provided to the Motorola 56001 code generator. MultiSim56000 is the code generation target. Some of the parameters used by this target are shown in window (2). The code generator partitions the algorithm and generates code for the two processors. In the resultant code, the first DSP performs linear distortion and phase shift operations, and the second adds Gaussian noise and second and third harmonic distortions. The hardware description for the system is developed using the SDF and Thor domains. The top-level design consisting of two processors and a shared memory is developed in the Thor domain (window (3)). To accommodate signals from both directions, a multiplexor/demultiplexor combination is added at the input and output of the first and the second processor respectively.

The analog components (A/D and D/A) are modeled in SDF (windows (4),(5), and (6)) and are added to the Thor universe as wormholes.

When this Thor Universe (window (3)) is run, the two DSPs run the code generated by the code generator (code0.asm and code1.asm). The transmitted and received (impaired) signals at both ends of the channel are observed and the design is verified.

This process may be repeated for different interprocessor communication mechanisms, or different hardware/software partitions. A thorough presentation of this case study can be found in [15], where both multiprocessor as well as system level design options are designed, and evaluated.

6.2 Programmable DSP core based ASIC Design

In Section 6.1 a codesign methodology for the design of multiprocessor systems was demonstrated. In this section the design approach for programmable DSP core-based ASIC designs is presented by stepping through the design of a modem.

6.2.1 Application Description and Specification

Figure 9 shows the transmitter and receiver for a modem [31]. We shall concentrate on the design of the receiver rather than the transmitter of the modem — it is usually the more challenging of the two. Channel equalization, carrier recovery, timing recovery, and symbol decoding are the critical components of the receiver.

6.2.2 System Design

Consider the design of a modem that would be embedded in a portable multimedia terminal — size, speed, and power are important considerations. Also, some programmability is necessary to allow changes in the constellation and fine tuning of the algorithm. In addition, the DSP used in the modem could also be used for other tasks in the terminal, such as front-end audio processing, fax, or voice mail. Thus, the design calls for high performance as well as programmability; hence, the programmable DSP core-based ASIC approach seems to be suitable for such a design.

A system consisting of two modems communicating over a channel is shown in the top window of Figure 10. The goal is to generate an implementation for the receiver section of this modem.

The algorithm for the receiver section is shown in the lower half of Figure 9. As a first cut in the design of the receiver section, the entire algorithm is mapped to software and is hence input to a single processor code generator (*Sim56*). On analyzing the generated code, it is found that it does not meet real-time requirements (the number of cycles between samples is less than the number of cycles required to process one sample). As a result, the algorithm is then partitioned so as to implement a part of the algorithm is custom hardware. The part of the algorithm to be implemented in hardware (say the AGC block) is next modeled in the Silage domain, and the part to be implemented as software is modeled in CG56. When the Silage model is run, Silage code corresponding to that part of the algorithm is generated. This Silage code is then passed to the Hyper synthesis tools that generate the layout and give estimates of the area and execution time. When the CG56 model is run, Motorola 56001 assembly code is generated for this part of the algorithm.

A hardware model for the receiver is next developed. The top-level description is in Thor and is shown in Figure 10. The signal from the channel is passed to an A/D interface (modeled in SDF). The digitized signal is passed to a Silage galaxy representing custom hardware. The internal structure of the Silage galaxy is shown on the left side of the figure. The "delay" block models the hardware execution delay (computed by the synthesis tools earlier). The signal from the custom hardware is transmitted to the serial port of the DSP. The rest of the algorithm is implemented as software running on the DSP. The code generated by the CG56 target (Sim56), is shown at the

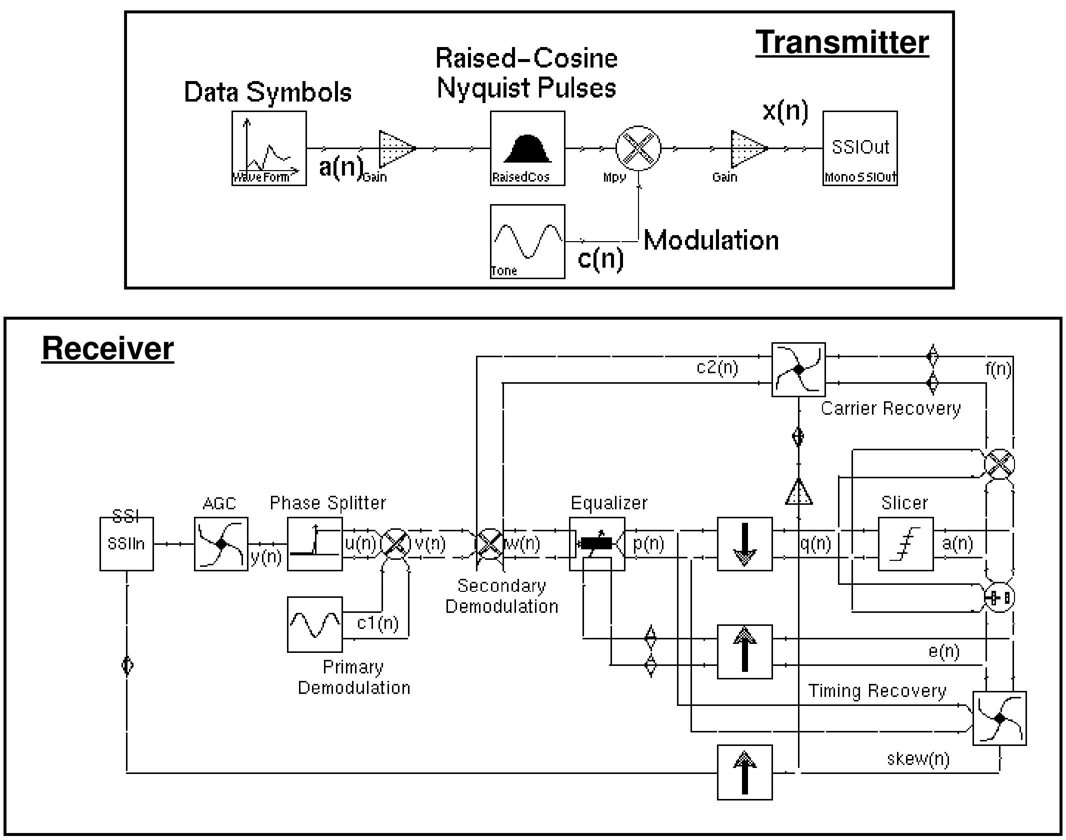

FIGURE 9. Transmitter (above) and Receiver (below) [31]

bottom right. The DSP and glue logic are implemented in Thor. The receiver can thus be implemented partly in software (as the program running on a programmable DSP) and partly in custom hardware (as a custom datapath). The system model can be simulated to verify functionality.

Using the DSP-core technology [20], this receiver can be synthesized as a single chip, where the custom datapath, DSP core, hardware/software interface circuitry, and glue logic can all be integrated into a single die. This integration of heterogeneous hardware/software structures results in improved performance, reduction of system size and power consumption, while still meeting programmability requirements.

Other options with different hardware/software partitions can be similarly constructed, and the design space can be explored.

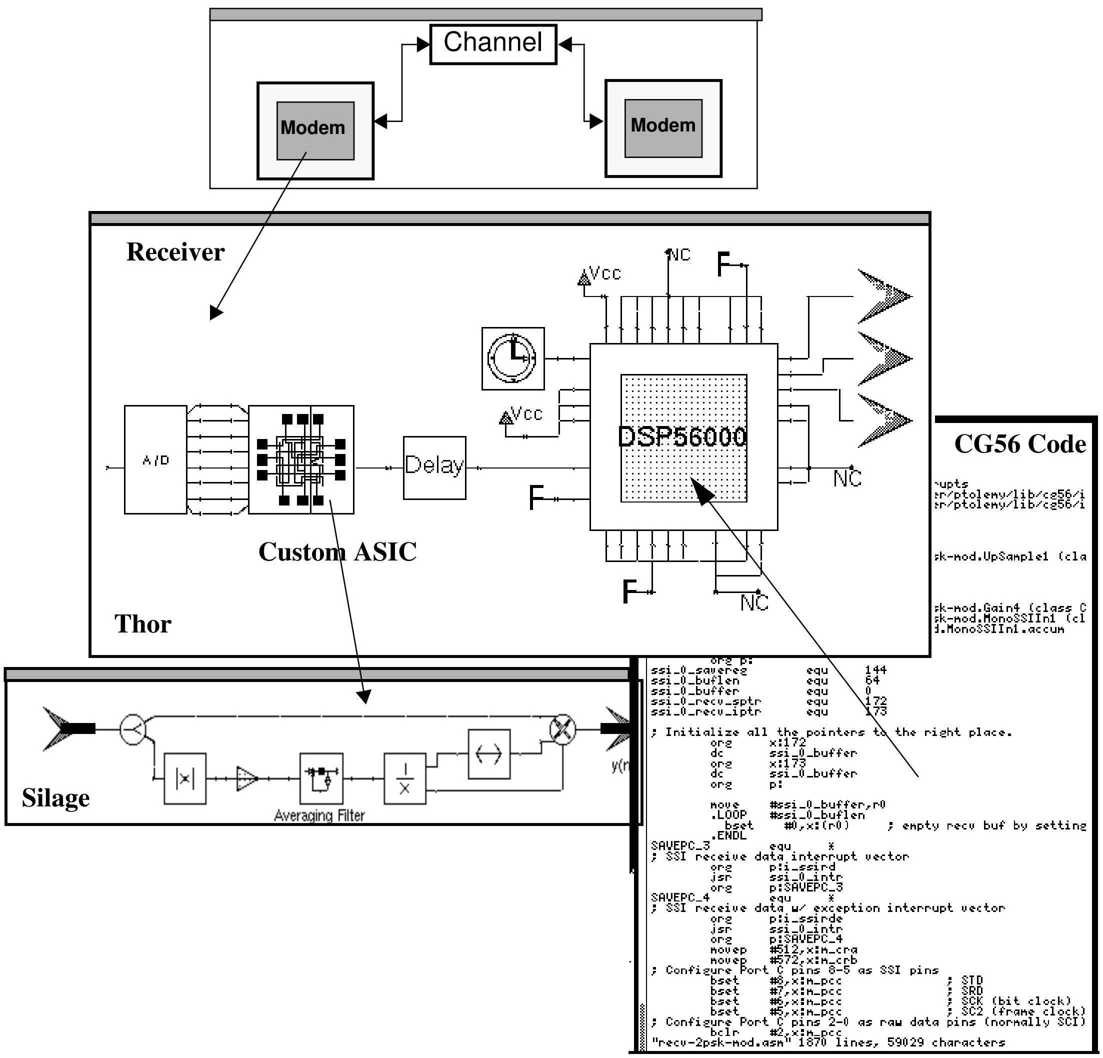

FIGURE 10. Modem design: The top window shows two modems communicating over a channel and the receiver configuration (Thor) is shown below it. The components implemented in custom hardware are modeled in Silage, and the DSP code is shown on the lower right.

7.0 Conclusions

The issues involved in the "codesign" of hardware and software have been presented in detail in the various sections of this paper. Though there is nothing new about designing systems containing both hardware and software, what really distinguishes the traditional design methodologies from the hardware/software codesign methodology that we have presented in this paper is *the integration of the two heterogeneous design methodologies within a single framework.* By having a unified representation for the hardware and the software, migration of functionality between the two implementations, and exploration of the design space becomes easily feasible. Our emphasis has not been on automating the process, but rather on providing the designer with advanced interoperable tools that enable creative exploration of the design space.

The codesign methodology under Ptolemy was demonstrated with the help of two examples. The design of the telephone channel simulator illustrates how different multiprocessor system configurations and hardware/software partitions can be explored fairly quickly and easily. The second example demonstrates the mechanism for combining synthesis of custom hardware with synthesis of assembly code for programmable components. The custom hardware and the programmable processor are integrated in a single simulation for the mixed hardware/software system.

Now that we have a functional infrastructure for doing hardware and software design, we are currently working on higher level tools that suport this methodology by developing a "Design Assistant" that would step the user through the various steps of the design, providing design advice and feedback at every stage. The Design Assistant is a special Target that co-ordinates the operation of the individual Targets in the hardware and the software domains. We are currently focusing on the design of systems (either at the board level, or as a DSP Core ASIC) containing a single processor and custom hardware. The methodology will next be extended for the design of heterogeneous systems containing multiple diverse processing elements.

Acknowledgments

The authors gratefully acknowledge the Semiconductor Research Corporation for supporting this work through its Berkeley Center of Excellence in CAD/IC (contract number 93-DC-008). Related work on the Ptolemy project has also been sponsored by DARPA, AT&T Bell Labs, Office of Naval Research, Bell Northern Research, Motorola, and Rockwell. The Ptolemy system has provided us with an extensive infrastructure of sophisticated tools such as code generators, parallel schedulers, and hardware simulators, plus the mechanisms to make them interoperate. Numerous individuals have contributed (and are contributing) to the development of the Ptolemy environment, and they are gratefully acknowledged: Shuvra Bhattacharyya, Joseph Buck, Wan-Teh Chang, Soonhoi Ha, Alan Kamas, Seungjun Lee, Prof. David G. Messerschmitt, Praveen Murthy, Thomas M. Parks, José Pino, and Kennard White.

8.0 References

[1] *"The Almagest: Manual for Ptolemy Version 0.4*", Department of EECS, University of California, Berkeley, CA 94720, USA, January 1993.

[2] S. Bhattacharyya, E. A. Lee, "Scheduling SDF Graphs for efficient looping", to appear in the *Journal of VLSI Signal Processing*, 1993.

[3] J. Bier and E. A. Lee, "Frigg: A Simulation Environment for Multiprocessor DSP System Development", *Proc. of Intl. Conf. on Computer Design*, Boston, MA, October, 1989.

[4] 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.

[5] J. Bier, S. Sriram, and E. A. Lee, "A Class of Multiprocessor Architectures for Real-Time DSP", *VLSI DSP IV*, ed. H. Moscovitz, IEEE Press, November, 1990.

[6] J. Bier, P. Lapsley, and E. A. Lee, "Tools and Methodologies for the Design of DSP Systems", Technical Report, Berkeley Design Technology, Inc., 38842 Helen Way, Fremont, CA 94536, 1993.

[7] J. T. Buck, "The Ptolemy Kernel", Memorandum UCB/ERL M93/8, 19 January 1993.

[8] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation,* special issue on "Simulation Software Development", to appear.

[9] R. Camposano, "From Behavior to Structure: Highlevel Synthesis", *IEEE Design and Test of Computers*, Oct. 1990, pp 8-19.

[10] P. Chou, R. Ortega, and G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-based systems", *Proc. of the Intl. Workshop on Hardware/Software Codesign*, Estes Park, Colorado, Sept. 1992.

[11] Electronic Industries Association Proposed Standard EIA-496-A Interface Between Data Circuit-Terminating Equipment (DCE) and the Public Switched Telephone Network (PSTN)", Prepared by EIA Technical Subcommittee TR-30.3.

[12] G. Goosens, F. Cathoor, D. Lanneer, and H. De Man, "Integration of Signal Processing Systems on heterogeneous IC architectures", *Proceedings of the Sixth International Workshop on High-Level Synthesis*, Laguna Niguel, CA, Nov. 1992.

[13] R. K. Gupta, G. De Micheli, "System-level Synthesis using Re-programmable Components", *Proceedings of the European Conference on Design Automation (EDAC-92)*, Brussels (Belgium), March 1992.

[14] P. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing", *Proceedings of the Custom Integrated Circuits Conference,* IEEE Computer Society Press, Los Alamitos, CA 1985, pp 213-216.

[15] A. Kalavade, "Hardware/Software Codesign using Ptolemy: A Case Study", Master's Report, EECS Dept., Univ. of California, Berkeley, Dec. 1991.

[16] A. Kalavade and E. A. Lee, "Hardware/Software Codesign Using Ptolemy: A Case Study", To appear in *Proceedings of the First IFIP Intl. Workshop on Hardware/Software Codesign*, Grassau, Germany, May 1992.

[17] E. A. Lee, D. G. Messerschmitt, "Digital Communications", Kluwer Academic Publishers, pp 128-135.

[18] E. A. Lee, D. G. Messerschmitt, "Synchronous Data Flow", *IEEE Proceedings,* September 1987.

[19] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. Bhattacharya, "Gabriel: A Design Environment for DSP", *IEEE Trans. on ASSP*, November, 1989.

[20] J. C. Lee, E. Cheval, and J. Gergen, "The Motorola 16-Bit DSP ASIC Core", *Proceedings of ICASSP 1990*, V3.12, pp 973-976.

[21] L. Mary, "DSP Based Technology for European Mobile Radio", Signal Processing V: Theory and Applications, *Proceedings of EUSIPCO-90,* Barcelona, Spain, September, 1990, pp 1495-1498.

[22] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "DSP Design Tool Requirements for the Nineties: An Industrial Perspective", submitted to the *Journal of VLSI Signal Processing,* special issue on "Synthesis for DSP".

[23] J. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software Synthesis for DSP Using Ptolemy", Submitted to the *Journal of VLSI Signal Processing,* special issue on "Synthesis for DSP".

[24] S. Prakash and A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", *Journal of Parallel and Distributed Computing,* 16, 338-351 (1992).

[25] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures", *IEEE Design and Test of Computers*, pp 40-51, June 1991.

[26] Rasure, and Williams, "An Integrated Visual Language and Software Development Environment", *Journal of Visual Languages and Computing*, Vol. 2, pp 217-246, 1991.

[27] G. C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication", Ph.D. Thesis, Electronic Research Laboratory, University of California, Berkeley, CA 94720, April 1991.

[28] M. B. Srivastava, "Rapid Prototyping of Hardware and Software in a Unified Framework", Ph.D. Thesis, University of California, Berkeley, CA, Memorandum No. UCB/ERL M92/67, June 1992.

[29] "*Thor Tutorial*", VLSI/CAD Group, Stanford University, 1986.

[30] I. Verbauwhede, J. M. Rabaey, "Synthesis of Real Time Systems: Solutions and Challenges", submitted to the *Journal of VLSI Signal Processing,* special issue on "Synthesis for DSP".

[31] K. White, "A 2PSK Passband Modem using Ptolemy", class project report, Fall 1992.

[32] Y. S. Wu, and L.J. Wu, "An Architectural Framework for Signal Flow", *Proceedings of the International Conference on Digital Signal Processing*, Florence, Sept. 1984.