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 baseclass 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*.

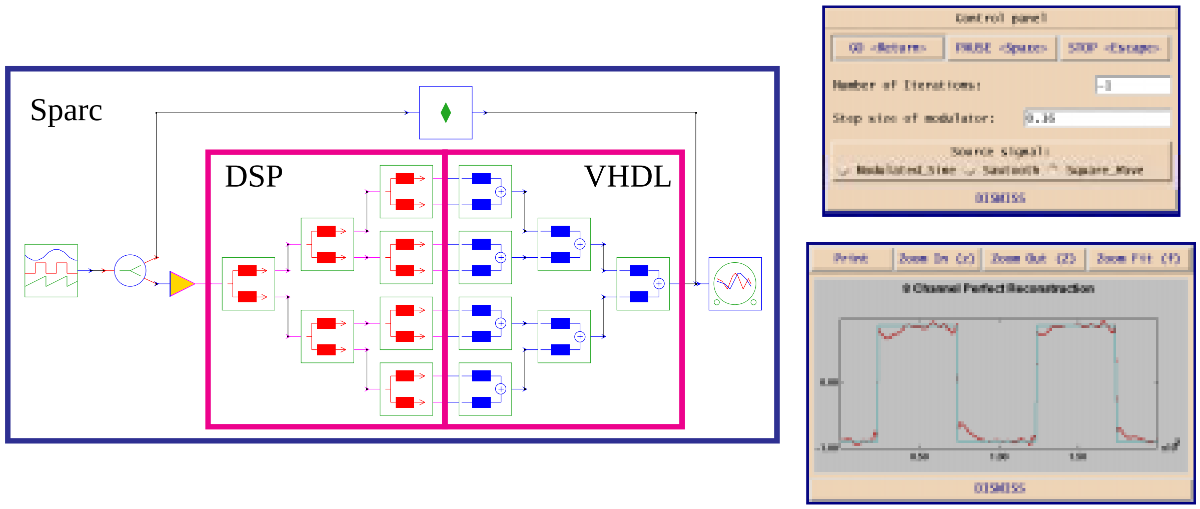
13.4.1 Interface Synthesis between Code Generation Targets

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

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 `EndCase`, 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

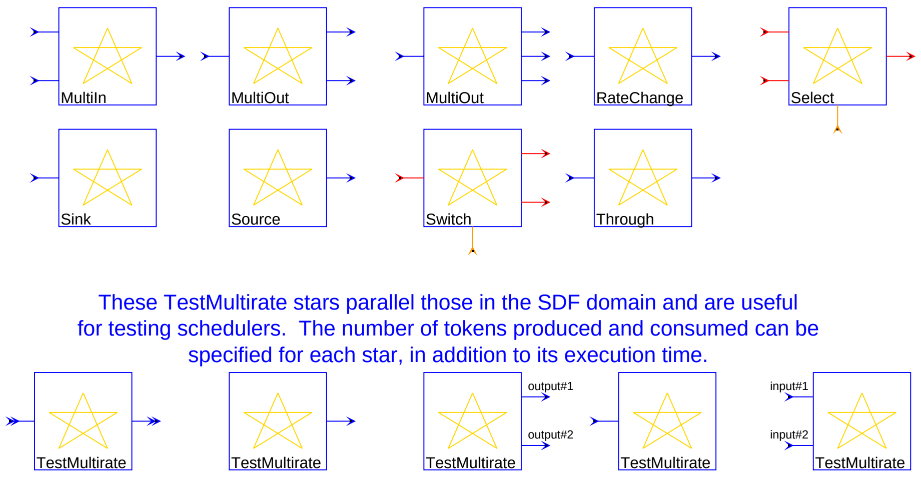

MultiIn
MultiOut
MultiOut
RateChange
Select
Sink
Source
Switch
Through
These TestMultirate stars parallel those in the SDF domain and are useful for testing schedulers. The number of tokens produced and consumed can be specified for each star, in addition to its execution time.
output#1
output#2
input#1
input#2
TestMultirate
TestMultirate
TestMultirate
TestMultirate
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.