Chapter 12. Code Generation

Authors: *Joseph Buck*
Soonhoi Ha
Edward A. Lee
Praveen K. Murthy
Thomas M. Parks
José Luis Pino
Kennard White

12.1 Introduction

The CG domain and derivative domains are used to generate code rather than to run simulations [Pin92]. Only the derivative domains are of practical use for generating code. The stars in the CG domain 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.

The CG domain is currently based on dataflow semantics. Dataflow models of computation in Ptolemy include synchronous dataflow (SDF), dynamic dataflow (DDF), and boolean dataflow (BDF). Both DDF and BDF are very general models of dataflow in that they are Turing equivalent. SDF is a subset of both these models. Hence, a code generation target that uses the BDF scheduler can support BDF and SDF stars but a target that uses SDF schedulers only supports SDF stars. Most targets in code generation domains use SDF schedulers and parallel schedulers which makes these targets support only SDF stars. An advantage of SDF is that compilation can be done statically; this permits very efficient code generation. While we have implemented targets that allow DDF code generation stars in the system, these targets are not in the current release. However, there are a couple of targets that use the BDF scheduler; refer to the BDF domain documentation, the section on the bdf-cg target in the CG domain documentation in the user's manual, and the section on the bdf-cgc target in the CGC domain documentation for more information on BDF semantics and the types of stars that can be supported. In this chapter, we assume that stars obey only SDF semantics since code generation for non-SDF models is still in its early stages.

The design goal of the code generation class hierarchy is to save work and to make the system more maintainable. Most of the work required to allocate memory for buffers, constants, tables, and to generate symbols that are required in code is completely processor-independent; hence these facilities are provided in the generic classes found in the `$PTOLEMY/src/domains/cg/kernel` directory.

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 it controls scheduling, multiprocessor architectures can be supported with automated task partitioning and synchronization.

In the following sections, we will introduce the methods and data structures needed to write new code generation stars and targets. However, we will not document what is needed to write a new code generation domain; that discussion can be found in chapter 15. We will first introduce what is needed to write a new code generation star, introducing the concepts of *code blocks*, *code streams* and *code block macros*. Next we will describe the various methods which will generally use the `addCode` method to piece together the code blocks into the code streams. We will then go into what is required to write single-processor and multiple-processor targets. Finally we will document the various schedulers available in the code generation domains.

12.2 Writing Code Generation Stars

Code generation stars are very similar to the C++ simulation stars. The main difference is that the initialization (`setup()`), run time (`go()`), and termination (`wrapup()`) methods generate code to be compiled and executed later. Additionally, code generation stars have two more methods called `initCode()` and `execTime()`.

The `setup()` method is called before the schedule is generated and before any memory is allocated. In this method, we usually initialize local variables or states. Note that the setup method of a star may be called multiple times. This means that the user should be careful so that the behavior of the star does not change even though setup method is called multiple times. The `initCode()` method of a star is called after the static schedule has been generated and before the schedule is fired. This method is used to generate the code outside of the main loop such as initialization code and procedure declaration code. To generate start-up code, use the `initCode` method, NOT the setup method, since setup is called before scheduling and memory allocation. The main use of the setup method, as in SDF, is to tell the scheduler if more than one sample is to be accessed from a porthole with the `setSDFParams` call.

The `go()` function is used to generate the main loop code for the star. Finally, the `wrapup()` function is used to generate the code after the main loop.

The `execTime()` method returns an integer specifying the time needed to execute the main loop code of a code generation star in processor cycles or instruction steps. These numbers are used by the parallel schedulers. In the assembly code generation domains, the integer returned is the main loop code execution time in DSP instruction cycles. The better the `execTime()` estimates are for each star, the more efficient the parallel schedule becomes.

If a star is invoked more than once during an iteration period, the precedence relation between stars should be known to the parallel scheduler. If there is no precedence relation between invocations, the parallel scheduler will try to parallelize them. By default, there is a precedence relation between invocations for any star (this is equivalent to having a self-loop). To assert that there is no such self-loop for a star, we have to call the `noInternalState()` method in the constructor:

```
constructor {
      noInternalState();
}
```

It is strongly recommended that the star designer determine whether the star is parallelizable or not, and call `noInternalState()` if it is.

The `CGStar` class is the base class for all code generation stars, such as high level language code generation stars and assembly language code generation stars. In this section, we will explain the common features that the `CGStar` class provides for all derivative code generation stars.

As a simple example to see how code generation stars are written, let's write an adder star for the C code generation domain. The `defstar` is almost the same as for a simulation star:

```
defstar {
      name {Add}
      domain {CGC}
      desc { Output the sum of the inputs, as a floating
value.}
      author { J. Pino }
      input {
            name {input1}
            type {float}
      }
      input {
            name {input2}
            type {float}
      }
      output {
            name {output}
            type {float}
      }
      ...
```

12.2.1 Codeblocks

Next we have to define the C code which will be used to generate the run-time code. For this we use a codeblock. A codeblock is a pseudo-language specification of a code segment. By pseudo-language we mean that the block of code is written in the target language with interspersed macros. Macros will be explained in the following section.

Codeblocks are implemented as protected static class members (e.g. there is one instance of a codeblock for the entire class). Since they are protected, codeblocks from one star class can be used from a derived star. The `codeblock` directive defines a block of code with an associated identifying name ("`addCB`" in this case).

```
codeblock (addCB) {
/* output = input1 + input2 */
$ref(output) = $ref(input1) + $ref(input2);
}
```

Special care should be given to codeblock specification. Within each line, spaces, tabs, and new line characters are important because they are preserved. For this reason, the brackets "`{ }`" should not be on the same lines with the code. Had `addCB` been defined as follows:

```
codeblock (addCB) { /* output = input1 + input2 */
$ref(output) = $ref(input1) + $ref(input2); }
```

the line

```
ref(output) = $ref(input1) + $ref(input2);
```

would be lost! This is because anything preceding the closing "`}`" on the same line is discarded by the preprocessor (`ptlang`). Secondly, the spaces and tabs between the opening "`{`" and the first non-space character will be ignored.

The first definition of the `addCB` codeblock is translated by `ptlang` into a definition of a static public member in the `.h` file:

```
class CGCAdd : public CGCStar
{
...
static CodeBlock addCB;
...
}
```

An associated constructor call will be generated in the .cc file:

```
CodeBlock CGCAdd :: addCB (
" /* output = input1 + input2 */\n"
" $ref(output) = $ref(input1) + $ref(input2);\n"
);
```

The argument is a single string, divided into lines for convenience. The following will complete our definition of the add star:

```
go {
     addCode(addCB);
}
```

Notice that the code is added in the go method, thus implying that the code is generated in the main loop.

The

```
addCode(code,stream name,<unique name>)
```

method of a CG star provides an interface to all the code streams (stream name and unique-name arguments are optional). This method defaults to adding code into the `myCode` stream (codestreams are explained later on). If a stream name is specified, `addCode` looks up the stream using the `getStream(`*`stream-name`*`)` method and then adds the code into that stream. Furthermore, if a unique name is provided for the code, the code will only be added if no other code has previously been added with the given unique name. The method `addCode` will return `TRUE` if the code-string has been added to the stream and otherwise will return `FALSE`.

The star just defined is a very simple star. Typical code generation stars will define many codeblocks. Conditional code generation is easily accomplished, as is done in the following example:

```
go {
     if (parameter == YES)
          addCode(yesblock);
     else
          addCode(noblock);
```

```
}
```

So far, we have used the `addCode()` method to generate the code inside the main loop body. In the assembly language domains, `addCode` can be called in the `initCode` and `wrapup` methods, to place code before or after the main loop respectively. In all of the code generation domains, we can use the `addProcedure()` method to generate declarations outside of the main body. Refer to "Code streams" on page 12-16 for documentation on the `addCode` and `addProcedure` methods.

The next section describes the extended codeblock support. The previous discussion of simple codeblocks is still correct and supported by `ptlang`; the extensions below are upward compatible. These extensions are experimental. They may change in future version of Ptolemy, and may still contain bugs.

12.2.2 Codeblocks with arguments

Simple codeblocks (as described above) have a name and are implemented as static member strings. Extended codeblocks have a name, optional arguments, and are implemented as non-static functions. They have an escape mechanism so that C++ expressions may be evaluated at run time and inserted into the generated code. An example:

```
codeblock(cbLoop,"int N, double x") {
      for (i=0; i < @N; i++) {
           $ref(output,i) = sin(i*@x);
      }
}
```

This defines a codeblock named `cbLoop` with two arguments: *N* and *x*. The variable *i* will appear in the generated code, while the C++ expressions *N* and *x* are escaped by `@` and will be evaluated at code-generation time. When this is called as

```
cbLoop(5, 0.1);
```

the following string will be returned:

```
      for (i=0; i < 5; i++) {
           $ref(output,i) = sin(i*0.1);
      }
```

This might be used within a `go()` method as:

```
go {
      addCode(cbLoop(5, 0.1));
}
```

The `addCode()` method will process the `$ref()` macro as described elsewhere. More complicated expressions are allowable. In general, the `@` clause may be delimited by parentheses "`(`"and "`)`", and must be operator << printable. The above codeblock could have been equivalently declared as:

```
    codeblock(cbLoop,"int N, double x") {
      for (i=0; i < @(N); i++) {
```

```
            $ref(output,i) = sin(i*@(x));
        }
    }
```

A more complicated example follows:

```
    codeblock(cbLoop2,"char *portname, int N, double x") {
        for (i=0; i < @(int(length)); i++) {
            $ref(@portname,i) = sin(i*@(x/N));
        }
    }
```

In this example, *length* is a data member of the star (typically a state). When called as:

```
        cbLoop2("ina", 3, 0.2);
```

it would generate (assuming the value of *length* is 20):

```
        for (i=0; i < 20; i++) {
            $ref(ina,i) = sin(i*0.6666666);
        }
```

In order to obtain the C++ expression processing via “@”-escapes in codeblocks without arguments, use a null argument list as in:

```
    codeblock(cbLoop3,"") {
        for (i=0; i < @(int(length)); i++) {
            $ref(output,i) = sin(i*0.1);
        }
    }
```

In the example above, the `@(int(length))` will be replaced with the value of the class member *length*. The above example would be called as:

```
    go {
        addCode(cbLoop3());
    }
```

The complete parsing rules are:

```
    @@          ==> @                   (double "@" goes to single)
    @ATSIGN     ==> @
    @{          ==> {
    @LBRACE     ==> {                   (LBRACE is literal string)
    @}          ==> }
    @RBRACE     ==> }                   (RBRACE is literal string)
    @\          ==> \
    @BACKSLASH ==> \                    (BACKSLASH is literal string)
    @id         ==> C++ token {id} (id is one or more alphanumerics)
    @(expr)     ==> C++ expr {expr}(expr is arbitrary with balanced
                parens)
```

```
@(white_space) ==>                nothing
@anything_else is passed through unchanged (including the @)
```

In an extended codeblock, trailing backslashes "\" will omit the following newline in the generated code. This special meaning of trailing "\" may be prevented by using "@\" or "@BACKSLASH".

12.2.3 In-line codeblocks

Code blocks may be specified in the body of a method. Inside the definition of a method (such as go()), all contiguous blocks of lines with a leading @ will be translated into an in-line codeblock (i.e., an addCode() statement). The @ escape mechanism for C++ expressions works as described above for codeblocks with arguments. Within @-escaped expressions, in-line codeblocks may reference local method variables as well as member variables.

Leading white-space before a leading @ will be ignored. Note that no override mechanism is provided to prevent the in-line codeblock interpretation. Note also that @ has dual meanings: the first @ on the line introduces in-line codeblock mode, while subsequent @ characters on the same line escape into C++ expressions. For example:

```
go() {
        @CMAM_wait( &$ref(ackFlag), 1);
}
```

is equivalent to:

```
go() {
        addCode("CMAM_wait( &$ref(ackFlag), 1);\n");
}
```

A more complicated example:

```
go {
    @    $ref(output) = \
    int ni = input.numberPorts();
    for (int i = 1; i <= ni; i++) {
      @$ref(input#@i) @(i < ni ? " + " : ";\n") \
    }
}
```

If "input.numberPorts()" returns 3 when the above program is run, the generated code will be:

```
"    $ref(output) = $ref(input#1) + $ref(input#2) + $ref(input#3);\n"
```

Currently, only the pre-defined methods (start, go, exectime etc.) are processed this way; not user-defined methods.

12.2.4 Macros

In code generation stars, the inputs and outputs no longer hold values, but instead correspond to target resources where values will be stored (for example, memory locations/registers in assembler generation, or global variables in C-code generation). A star writer can also define states which can specify the need for global resources.

A code generation star, however, does not have knowledge of the available global resources or the global variables/tables which have already been defined in the generated code. For star writers, a set of macros to access the global resources is provided. The macros are expanded in a language or target specific manner after the target has allocated the resources properly. In this section, we discuss the macros defined in the `CGStar` class.

`$ref(name)`
: Returns a reference to a state or a port. If the argument, name, refers to a port, it is functionally equivalent to the `name%0` operator in the SDF simulation stars. If a star has a multi-porthole, say *input*, the first real porthole is *input#1*. To access the first porthole, we use `$ref(input#1)` or `$ref(input#internal_state)` where `internal_state` is the name of a state that has the current value, 1.

`$ref(name,offset)`
: Returns a reference to an array state or a port with an offset that is not negative. For a port, it is functionally equivalent to `name%offset` in SDF simulation stars.

`$val(state-name)`
: Returns the current value of the state. If the state is an array state, the macro will return a string of all the elements of the array spaced by the new line character. The advantage of not using `$ref` macro in place of `$val` is that no additional target resources need to be allocated.

`$size(name)`
: Returns the size of the state/port argument. The size of a non-array state is one; the size of a array state is the total number of elements in the array. The size of a port is the buffer size allocated to the port. The buffer size is usually larger than the number of tokens consumed or produced through that port.

`$starName()`
: Returns the instantiated name of the star (without galaxy or universe names)

`$fullName()`
: Returns the complete name of the star including the galaxies to which it belongs.

`$starSymbol(name)`
: Returns a unique label in the star instance scope. The instance scope is owned by a particular instance of that star in a graph. Furthermore, the scope is alive across all firings of that particular star. For example, two CG stars will have two distinct star instance scopes. As an example, we show some parts of ptlang

file of the `CGCPrinter` star.

```
initCode {
...
      StringList s;
      s << " FILE* $starSymbol(fp);";
      addDeclaration(s);
      addInclude("<stdio.h>");
      addCode(openfile);
...
}
codeblock (openfile) {
      if(!($starSymbol(fp)=fopen("$val(fileName)","w"))) {
            fprintf(stderr,"ERROR: cannot open output file
for Printer star.\n");
      exit(1);
      }
}
```

The file pointer `fp` for a star instance should be unique globally, and the `$starSymbol` macro guarantees the uniqueness. Within the same star instance, the macro returns the same label.

`$sharedSymbol(list,name)`

Returns the symbol for name in the list scope. This macro is provided so that various stars in the graph can share the same data structures such as sin/cos lookup tables and conversion table from linear to mu-law PCM encoder. These global data structures should be created and initialized once in the generated code. The macro `sharedSymbol` does not provide the method to generate the code, but does provide the method to create a label for the code. To generate the code only once, refer to “Code streams” on page 12-16. A example where a shared symbol is used is in `CGCPCM` star.

```
codeblock (sharedDeclarations)
{
      int $sharedSymbol(PCM,offset)[8];
      /* Convert from linear to mu-law */
      int $sharedSymbol(PCM,mulaw)(x)
      double x;
      {
            double m;
            m = (pow(256.0,fabs(x)) - 1.0) / 255.0;
            return 4080.0 * m;
      }
}
codeblock (sharedInit)
{
      /* Initialize PCM offset table. */
      {
      int i;
      double x = 0.0;
      double dx = 0.125;
      for(i = 0; i < 8; i++, x += dx)
      {
            $sharedSymbol(PCM,offset)[i] =
                              $sharedSymbol(PCM,mulaw)(x);
      }
}
initCode {
...
      if (addGlobal(sharedDeclarations, "$sharedSym-
bol(PCM,PCM)"))
      addCode(sharedInit);
}
```

The above code creates a conversion table and a conversion function from linear to mu-law PCM encoder. The conversion table is named `offset` and belongs to the `PCM` class. The conversion function is named `mulaw`, and belongs to the same PCM class. Other stars can access that table or function by saying `$sharedSymbol(PCM,offset)` or `$sharedSymbol(PCM,mulaw)`. The `initCode` method tries to put the `sharedDeclarations` codeblock into the global scope (by `addGlobal()` method in the CGC domain). That code block is given a unique label by `$sharedSymbol(PCM,PCM)`. If the codeblock has not been previously defined, `addGlobal` returns true, thus allowing `addCode(sharedInit)`. If there is more than one instance of the PCM star, only one instance will succeed in adding the code.

`$label(name), $codeblockSymbol(name)`

Returns a unique symbol in the codeblock scope. Both label and codeblockSymbol refer to the same macro expansion. The codeblock scope only lives as long as a codeblock is having code generated from it. Thus if a star uses `addCode()` more than once on a particular codeblock, all codeblock instances will have unique symbols. A example of where this is used in the `CG56HostOut` star.

```
codeblock(cbSingleBlocking) {
$label(wait)
jclr #m_htde,x:m_hsr,$label(wait)
jclr #0,x:m_pbddr,$label(wait)
movep $ref(input),x:m_htx
}
codeblock(cbMultiBlocking) {
move #$addr(input),r0
.LOOP #$val(samplesOutput)
$label(wait)
jclr #m_htde,x:m_hsr,$label(wait)
jclr #0,x:m_pbddr,$label(wait)
movep x:(r0)+,x:m_htx
.ENDL
nop
}
```

The above two codeblocks use a label named *wait*. The `$label` macro will assign unique strings for each codeblock.

The base `CGStar` class provides the above 8 macros. In the derived classes, we can add more macros, or redefine the meaning of these macros. Refer to each domain document to see how these macros are actually expanded. There are three commonly used macros in the assembly code generation domains; these are:

`$addr(name)`
: This returns the address of the allocated memory location for the given state or porthole name. The address does not include references to the memory bank the location is coming from; for instance, “x:2034” for location 2034 in the “x” memory bank for Motorola 56000 is output as 2034.

`$addr(name,<offset>)`
: This macro returns the numeric address in memory of the named object, without (for the 56000) an “x:” or “y:” prefix. If the given quantity is allocated in a register (not yet supported) this function returns an error. It is also an error if the argument is undefined or is a state that is not assigned to memory (e.g. a parameter).
Note that this does NOT necessarily return the address of the beginning of a porthole buffer; it returns the “access point” to be used by this star invocation, and in cases where the star is fired multiple times, this will typically be different from execution to execution.
If the optional argument offset is specified, the macro returns an expression that references the location at the specified offset -- wrapping around to the beginning of the buffer if that is necessary. Note that this wrapping works independent of whether the buffer is circularly aligned or not.

`$ref(name,<offset>)`

This macro is much like $addr(name), only the full expression used to refer to this object is returned, e.g. “x:23” for a 56000 if “name” is in x memory. If “name” is assigned to a register, this expression will return the corresponding register. The error conditions are the same as for $addr

`$mem(name)`
: Returns the name of the memory bank in which the given state or porthole has its memory allocated.

To have “`$`” appear in the output code, put “`$$`” in the codeblock. For a domain where “`$`” is a frequently used character in the target language, it is possible to use a different character instead by redefining the virtual function `substChar` (defined in `CGStar`) to return a different character.

It is also possible to introduce processor-specific macros, by overriding the virtual function `processMacro` (rooted in `CGStar`) to process any macros it recognizes and defer substitution on the rest by calling its parent’s `processMacro` method.

12.2.5 Assembly PortHoles

Here are some methods of class `AsmPortHole` that might be useful in assembly code generation stars:

`bufSize()` Returns an integer, the size of the buffer associated with the porthole.

`baseAddr()` Returns the base address of the porthole buffer

`bufPos()` Returns the offset position in the buffer, which ranges from 0 to `bufSize()`-1.

`circAccessThisTime()`
: This method returns true (nonzero) if the data to be read or written on this execution “wrap around”, so that accessing them in a linear order will not work.

12.2.6 Attributes

Attributes are assertions about the object they are applied to. Both states and portholes can have attributes. Attributes that apply to states have the prefix “`A_`”. Attributes that apply to portholes have the prefix “`P_`”. The following attributes are common to all code generation domains:

`A_GLOBAL`
: If set, this state is declared global so that it is accessible everywhere. Currently, it is only supported in the CGC domain.

`A_LOCAL`
: This is the opposite of `A_GLOBAL`.

`A_SHARED`
: A state that is shared among all stars that know its name, type, size.

`A_PRIVATE`

Opposite of `A_SHARED`.

The default for stars is `A_LOCAL|A_PRIVATE`. Right now, only `A_SHARED|A_LOCAL` is supported in the assembly language domains. This combination means that all stars will share the particular state across a processor. For all stars to share it in a universe the bits `A_SHARED|A_GLOBAL` need to be set; this combination is not implemented yet - the default method will probably restrict all the stars that share this state to the same processor.

`A_CONSTANT`
: The state value is not changed by the star's execution.

`A_NONCONSTANT`
: The state value is changed by the star's execution.

`A_SETTABLE`
: The user may set the value of this state from a user interface.

`A_NONSETTABLE`
: The user may not set the value of this state from a user interface (e.g. edit-parameters doesn't show it).

Applying an attribute to an object implies that some bits are to be "turned on", and others are to be "turned off". The underlying attribute bits have names beginning with `AB_` for states, and `PB_` for portholes. The only two bits that exist in all states are `AB_CONST` and `AB_SETTABLE`. By default, they are on for states, which means that the default state works like a parameter (you can set it from the user interface, and the star's execution does not change it).

For assembly language domains, the following attributes are defined:

`A_CIRC`
: If set, the memory for this state is allocated as a circular buffer, whose address is aligned to the next power of two greater than or equal to its length.

`A_CONSEC`
: If set, allocate the memory for the *next* state in this star consecutively, starting immediately after the memory for this star.

`A_MEMORY`
: If set, memory is allocated for this state.

`A_NOINIT`
: If set, the state is not be automatically initialized. The default is that all states that occupy memory are initialized to their default values.

`A_REVERSE`
: If set, write out the values for this state in reverse order.

`A_SYMMETRIC`
: If set, and if the target has dual data memory banks (e.g. M56000, Analog Devices 2100, etc.), allocate a buffer for this object in both memories.

Given these attributes (technically, the above also have "bit" representations of the form AB_xxx; A_xxx just turns the bit AB_xxx on), the following attributes correspond to requests to turn some attributes off and to turn other attributes on. For example:

`A_ROM`
: Allocate memory for this state in memory, and the value will not change -- `A_MEMORY` and `A_CONSTANT` set.

`A_RAM`
: `A_MEMORY` set, `A_CONST` not set

For portholes in code generation stars, we have:

`P_CIRC`
: If set, then allocate the buffer for this porthole as a circular buffer, even if this is not required because of any other consideration.

`P_SHARED`
: Equivalent to `A_SHARED`, only for portholes.

`P_SYMMETRIC`
: Similar to `A_SYMMETRIC`, but for portholes.

`P_NOINIT`
: Do not initialize this porthole.

Attributes can be combined with the "|" operator. For example, to allocate memory for a state but make it non-settable by the user, I can say
`AB_MEMORY|A_NONSETTABLE`

12.2.7 Possibilities for effective buffering

In principle, blocks communicate with each other through porthole connections. In code generation domains, we allocate a buffer for each input-output connection by default. There are some stars, however, that do not modify data at all. A good, and also ubiquitous, example is a `Fork` star. When a `Fork` star has *N* outputs, the default behavior is to create *N* buffers for output connections and copy data from input buffer to *N* output buffers, which is a very expensive and silly approach. Therefore, we pay special attention to stars displaying this type of behavior. In the setup method of these stars, the `forkInit()` method is invoked to indicate that the star is a `Fork`-type star. For example, the `CGCFork` star is defined as

```
defstar {
name { Fork }
domain { CGC }
desc { Copy input to all outputs }
version { @(#)CGCFork.pl 1.6 11/11/92 }
author { E. A. Lee }
copyright { 1991-1994 The Regents of the University of Cali-
fornia }
location { CGC demo library }
explanation {
```

```
Each input is copied to every output. This is done by the way
the buffers are laid out; no code is required.
}
input {
      name {input}
      type {ANYTYPE}
}
outmulti {
      name {output}
      type {=input}
}
constructor {
      noInternalState();
}
start {
      forkInit(input,output);
}
exectime { return 0;}
}
```

Where possible, code generation domains take advantage of `Fork`-type stars by not allocating output buffers, but instead the stars reuse the input buffers. Unfortunately, in the current implementation, assembly language fork stars can not do their magic if the buffer size gets too large (specifically, if the size of the buffer that must be allocated is greater than the total number of tokens generated or read by some port during the entire execution of the schedule). Here, forks or delay stars that copy inputs to outputs must be used.

Another example of a `Fork`-Type star is the `Spread` star. The star receives *N* tokens and spreads them to more than one destination. Thus, each output buffer may share a subset of its input buffer. We call this relationship *embedding*: the outputs are embedded in the input. For example, in the `CGCSpread` star:

```
setup {
      MPHIter iter(output);
      CGCPortHole* p;
      int loc = 0;
      while ((p = (CGCPortHole*) iter++) != 0) {
            input.embed(*p, loc);
            loc += p->numXfer();
      }
}
```

Notice that the output is a multi-porthole. During setup, we express how each output is embedded in the input starting at location *loc*. At the buffer allocation stage, we do not allocate buffers for the outputs, but instead reuse the input buffer for all outputs. This feature, however, has not yet been implemented in the assembly language generation domains.

A `Collect` star embeds its inputs in its output buffer:

```
setup {
      MPHIter iter(input);
```

```
            CGCPortHole* p;
            int loc = 0;
            while ((p = (CGCPortHole*) iter++) != 0) {
                  output.embed(*p, loc);
                  loc += p->numXfer();
            }
      }
```

Other examples of embedded relationships are `UpSample` and `DownSample` stars. One restriction of embedding, however, is that the embedded buffer must be static. Automatic insertion of `Spread` and `Collect` stars in multi-processor targets (refer to the target section) guarantees static buffering. If there is no delay (i.e., no initial token) in the embedded buffer, static buffering is enforced by default. A buffer is called *static* when a star instance consumes or produces data in the same buffer location in any schedule period. Static buffering requires a size that divides the least common multiple of the number of tokens consumed and produced; if such a size exists that equals or exceeds the maximum number of data values that will ever be in the buffer, static allocation is performed.

12.3 Targets

A code generation **Domain** is specific to the language generated, such as C (CGC), Sproc assembly code (Sproc) [Mur93], Silage [Kal93], DSP56000 assembly code (CG56), and DSP96000 assembly code (CG96). Each code generation domain has a default target which defines routines generic to the target language. A derived `Target` that defines architecture specific routines can then be written. A given language, particularly a generic language such as C, may run on many target architectures. Code generation functions are cleanly divided between the default domain target and the architecture specific target.

All target architectures are derived from the base class `Target`. The special class `KnownTarget` is used to add targets to the known list of targets, much as `KnownBlock` is used to add stars (and other blocks) to the known block list and to assign names to them.

A `Target` object has methods for generating a schedule, compiling the code, and running the code (which may involve downloading code to 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.

12.3.1 Single-processor target

The base target for all code generation domains is the `CGTarget`, which represents a single processor by default. This target is called *default-CG* in the target list for the CG domain. As the generic code generation target, the `CGTarget` class defines many common functions for code generation targets. Methods defined here include virtual methods to generate, display, compile, and run the code. Derived targets are free to redefine these virtual methods if necessary.

Code streams

A code generation target manages code streams which are used to store star and target generated code. The `CGTarget` class has the two predefined code streams: `myCode` and `pro-`

`cedures`. The `myCode` stream is referred to as `CODE` and the `procedures` stream is called `PROCEDURE`; these names should be used when referring to these streams as in "`CodeStream* code = getStream(CODE)`". Derived targets are free to add more code streams using the `CGTarget` method `addStream(`*`stream-name`*`)`. For example, the default CGC target defines fourteen additional code streams.

Other methods, such as `addProcedure(code,`*`uniquename`*`)` can be defined, to provide a more efficient or convenient interface to a specific code stream (in this case, procedures). With `addProcedure` it becomes clear why unique names are necessary. Recall that `addProcedure` is used to declarations outside of the main body of the code. For example, say we wanted to write a function in C to multiply two numbers. The codeblock to do this could read:

```
codeblock(sillyMultiply) {
/* A silly function */
double $sharedSymbol(silly,mult)(double a, double b)
{
      double m;
      m = a*b;
      return m;
}
}
```

Note that in this codeblock we used the `sharedSymbol` macro described in the code generation macros section. To add this code to the procedures stream, in the `initCode` method of the star, we can call either:

```
addProcedure(sillyMultiply,"mult");
```

or

```
addCode(sillyMultiply,"procedures","mult");
```

or

```
getStream("procedures")->put(sillyMultiply,"mult");
```

As with `addCode`, `addProcedure` returns a `TRUE` or `FALSE` indicating whether the code was inserted into the code stream. Taking this into account, we could have added the code line by line:

```
if (addProcedure("/* A silly function */\n","mult")) {
      addProcedure(
      "double $sharedSymbol(silly,mult)(double a, double
b)\n"
      );
      addProcedure("{\n");
      addProcedure("\tdouble m;\n");
      addProcedure("\tm = a*b;\n");
      addProcedure("\treturn m;\n");
      addProcedure("}\n");
}
```

12.3.2 Assembly code streams

Code is generated in the assembly language domains into four streams. The streams inherited from `CGTarget` are the `CODE` and `PROCEDURES` stream. The two new streams are:

`mainLoop` Code added to this stream comprises the main loop of the generated algorithm. All `addCode` calls from a star's `go` function automatically are concatenated to this stream unless another stream is supplied as an argument.

`trailer` Code added to this stream comprises the `wrapup` section of the generated algorithm. All `addCode` calls from a star's `wrapup` method automatically are concatenated to this stream unless another stream is supplied as an argument.

Code generation

Once the program graph is scheduled, the target generates the code in the virtual method `generateCode()`. (Note: code streams should be initialized before this method is called.) All the methods called by `generateCode` are virtual, thus allowing for target customization. The `generateCode` method then calls `allocateMemory()` which allocates the target resources. After resources are allocated, the `initCode` method of the stars are called by `codeGenInit()`. The next step is to form the main loop by calling the method `mainLoopCode()`. The number of iteration cycles are determined by the argument of the "run" directive which a user specifies in `pigi` or in `ptcl`. To complete the body of the main loop, `go()` methods of stars are called in the scheduled order. After forming the main loop, the `wrapup()` methods of stars are called.

Now, all of the code has been generated; however, the code can be in multiple target streams. The `frameCode()` method is then called to piece the code streams together and place the unified stream into the `myCode` stream. Finally, the code is written to a file by the method `writeCode()`. The default file name is *"code.output"*, and that file will be located in the directory specified by a target parameter, *destDirectory*.

Finally, since all of the code has been generated for a target, we are ready to compile, load, and execute the code. Derived targets should redefine the virtual methods `compileCode()`, `loadCode()`, and `runCode()` to do these operations. At times it does not make sense to have separate `loadCode()` and `runCode()` methods, and in these cases, these operations should be collapsed into the `runCode()` method.

12.3.3 Multiprocessor targets

Targets representing multiple processors are derived from the `CGTarget` class. The base class for all multiple-processor targets is called `MultiTarget`, and resides in the `$(PTOLEMY)/src/domains/cg/kernel` directory. `CGMultiTarget` is derived from `MultiTarget`. `CGMultiTarget` class is the base class for all multiple-processor targets. It is called *FullyConnected* in the CG domain target list.

The design of Ptolemy is also intended to support heterogeneous multi-processor targets. In the future, the base class of all "abstract" heterogeneous multiprocessor targets will be implemented from the `MultiTarget` class. For such targets, certain actors must be assigned to certain targets, and the cost of a given actor is in general a function of which child target it is assigned to. We have developed parallel schedulers that address this problem [Sih91].

We have implemented, or are in the process of implementing, both "abstract" and "concrete" multi-processor targets. For example, we have classes named `CGMultiTarget` and `CGSharedBus` that represent sets of homogenous single-processor targets of arbitrary type, connected in either a fully connected or shared-bus topology, with parametrized communication costs. These targets, however, use only the CG domain stars and hence do not actually generate code (recall that CG domain stars are "comment generators"). Some other actual implementations of multiprocessor systems include the CM-5 (`CGCCm5Target` in the CGC domain), and the ordered transaction architecture [Sri93]. Refer to the CG56 domain documentation for `CG56MultiSim` target, or the CGC domain documentation for `CGCMultiTarget` class as examples of "concrete" multi-processor targets. In this section, we concentrate on the "abstract" multiprocessor target classes that are in the `$(PTOLEMY)/src/domains/cg/targets` directory.

`CGMultiTarget` is the base target class for all homogeneous targets. By default, it models a fully-connected multiprocessor architecture; when a processor wants to communicate with another processor, it can do immediately. The `scheduleComm()` method returns the time when the required communication is scheduled. In the `CGMultiTarget` class, it returns the same time as when the communication is required. On the other hand, `CGSharedBus`, which is derived from the `CGMultiTarget` class, is the base target class for all multiprocessor targets having a shared-bus topology. In the `CGSharedBus` class, the `scheduleComm()` method schedules the required communication on the shared-bus member object of that class, and returns the scheduled time. The communication cost (in time) is modeled by the `commTime()` method. Given the information on which processors are involved in this communication and how many tokens are transmitted, it returns the expected communication time once started. By default (or in fully-connected topology), it only depends on the number of tokens.

A `CGMultiTarget` has a sequence of child target objects to represent each of the individual processors. The number of processors are determined by an `IntState`, `nprocs`, and the type of the child target is specified by a `StringState`, `childType`. Refer to the *User's Manual* for details on how to specify the various target parameters. In the setup stage, the child targets are created and added to the child target list as members of the multiprocessor target. Classes derived from `MultiTarget` represent the topology of the multi-processor network (communication costs between processors, schedules for use of communication facilities, etc.), and single-processor child targets can represent arbitrary types of processors. The resource allocation problem is divided between the parent target, representing the shared resources, and the child targets, representing the resources that are local to each processor.

The main role of a multiprocessor target is to set up one of the chosen parallel schedulers, and to coordinate the child targets. The `CGMultiTarget` class has a set of parameters to select parallel scheduling options. See the schedulers section for a detailed discussion on parallel schedulers. The selected parallel scheduler schedules the program graph onto the child targets and the scheduling results are displayed on a Gantt chart. The parent multiprocessor target collects the code from each of the child targets after the child targets have generated code based on the scheduling results. By default, it merges all of the child-processor code into a single file. If separate files are required, then one approach is to create separate files with names derived from the child target names and write the code to these files in the `frameCode()` method of the multi-target.

Interprocessor communication (IPC) stars are created by the multiprocessor target by the methods `createSend()` and `createReceive()`. These stars are spliced in to the sub-galaxies that are created and handed down to the child targets. Typically, these methods just create the appropriate IPC star and return a pointer to the object created. Each send/receive pair is matched in the `pairSendReceive()` method. Typically, this might involve setting pointers in the send/receive pair to point to each other.

There is no preprocessor for targets like `ptlang` for stars. Designing a customized multiprocessor target, therefore, is a bit complicated compared to designing a customized star. If the interconnection topology is neither fully-connected nor shared-bus, in particular, the communication scheduling should be designed in the target, which makes a target design more complicated. So the best way to design a target is to look at an already-implemented target such as `CGCMultiTarget` class in the CGC domain.

12.4 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.

In this section, we explain different scheduling options and their effect on the generated code.

12.4.1 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` = 0), 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. 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.

We call the second approach *Joe's* scheduler. In this approach (`loopingLevel` = 1), 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 2, we can choose the third approach, called *SJS* (Shuvra-Joe-Soonhoi) scheduling after the inventor's first names [Bha94]. 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.

If the second or third approach is 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 (runtime overhead) for indirect addressing.

12.4.2 Multiprocessor schedulers

A key idea in Ptolemy is that there is no single scheduler that is expected to handle all situations. Users can write schedulers and can use them in conjunction with schedulers we have written. As with the rest of Ptolemy, schedulers are written following object-oriented design principles. Thus a user would never have to write a scheduler from ground up, and in fact the user is free to derive the new scheduler from even our most advanced schedulers. We have designed a suite of specialized schedulers that can be mixed and matched for specific applications.

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.

We have implemented three scheduling techniques that map SDF graphs onto multiple-processors with various interconnection topologies: Hu's level-based list scheduling, Sih's dynamic level scheduling [Sih91], and Sih's declustering scheduling [Sih91]. The target architecture is described by its `Target` object, derived from `CGMultiTarget`. The Target class provides the scheduler with the necessary information on interprocessor communication to enable both scheduling and code synthesis.

The `CGMultiTarget` has a parameter, *schedName*, that allows the user to select the type of schedule. Currently, there are five different scheduling options:

`DL` — If *schedName* is set to `DL`, we select the Sih's dynamic level scheduler that accounts for IPC overhead during scheduling.

`HU` — Hu's level scheduler is selected, which ignores the IPC overhead.

`DC` The Sih's declustering scheduler can be selected by setting `DC`. The declustering algorithm is advantageous only when the list scheduling algorithm shows poor performance, judged from the scheduling result because it is more expensive than the `DL` or `HU` scheduler.

`HIER(DL)` or `HIER(HU)` or `HIER(DC)`
If we want to use Pino's hierarchical scheduler, we have to set *schedName* to `HIER(DL` or `HU` or `DC)`. The default top-level scheduling option is the `DL` scheduler. To use other scheduler, `DC` or `HU` should be specified within the parenthesis.

`CGDDF` If the *schedName* is set to `CGDDF`, the Ha's dynamic construct scheduler is selected. To use this scheduler, Ptolemy should be recompiled with special flags, or use `mkcgddf` executable.

Whichever scheduler is used, we schedule communication nodes in the generated code. For example, if we use the Hu's level-based list scheduler, we ignore communication overhead when assigning stars to processors. Hence, the code is likely to contain more communication stars than with the other schedulers that do not ignore 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` option automatically sets the `oneStarOneProc` state to be 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 which 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 setting that state. The `adjustSchedule` cannot be `YES` before any scheduling decision is made previously. Again, this option is not supported in this release.

Different scheduling options result in different assignments of APEG nodes. 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 multiple-processor scheduler produces a list of single processor schedules, giving them to the child targets. The schedules include send/receive stars for interprocessor communication. The child targets take their schedules and generate code.

To produce code for child targets, we create a sub-galaxy for each child target, which consists of the stars scheduled on that target and some extra stars to be discussed below if necessary. A child target follows the same step to generate code as a single processor target except that the schedule is not computed again since the scheduling result is inherited from the parent target.

Send/Receive stars

After the assignment of APEG nodes is finished, the interprocessor communication requirements between blocks are determined in sub-galaxies. Suppose star A is connected to star B, and there is no sample rate change. By assigning star A and star B to different processors (1 and 2 respectively), the parallel scheduler introduces interprocessor communication. Then, processor 1 should generate code for star A and a “send” star, while processor 2 should generate code for a “receive” star and star B. These “send” and “receive” stars are inserted automatically by the Ptolemy kernel when determining the execution order of blocks in each child target and creating the sub-galaxies. The actual creation of send/receive stars is done by the parallel scheduler by invoking methods (`createSend()` and `createReceive()`, as mentioned earlier) in the parent multi-target.

Once the generated code is loaded, processors run autonomously. The synchronization protocol between processors is hardwired into the “send” and “receive” stars. One common approach in shared-memory architectures is the use of semaphores. Thus a typical synchronization protocol is to have the send star set a flag when it completes the data transfer, and have the receive star read the data and reset the semaphore. The receive star will not read the data if the semaphore has not been set and similarly, the send star will not write data if the semaphore has not been reset. In a message passing architecture, the send star may form a message header to specify the source and destination processors. In this case, the receive star would decode the message by examining the message header.

For properly supporting arbitrary data types, the send star should have an `ANYTYPE` input; the receive star should have an `ANYTYPE` output. The resolved type for each of these ports can be obtained using the `Porthole::resolvedType` method. For a preliminary version of the communication stars, you can use a fixed datatype such as `FLOAT` or `INT`.

The send/receive stars that are declared to support `ANYTYPE` but fail to support a particular datatype, should display an appropriate error message using the `Error::abortRun` method. Finally, each of these stars must call `PortHole::numXfer` to determine the size of the block of data that needs to be transferred upon each invocation.

Spread/Collect stars

Consider a multi-rate example in which star A produces two tokens and star B consumes one token each time. Suppose that the first invocation of star B is assigned to the same processor as the star A (processor 1), but the second invocation is assigned to processor 2. After star A fires in processor 1, the first token produced should be routed to star B assigned to the same processor while the second token produced should be shipped to processor 2; interprocessor communication is required! Since star A has one output port and that port should be connected to two different destinations (one is to star B, the other is to a "send" star), we insert a "spread" star after star A. As a result, the sub-galaxy created for processor 1 contains 4 blocks: star A is connected to a "spread" star, which in turn has two outputs connected to star B and a "send" star. The role of a "spread" star is to spread tokens from a single output porthole to multiple destinations.

On the other hand, we may need to "collect" tokens from multiple sources to a single input porthole. Suppose we reverse the connections in the above example: star B produces one token and star A consumes two tokens. We have to insert a "collect" star at the input porthole of star A to collect tokens from star B and a "receive" star that receives a token from processor 2.

The "spread" and "collect" stars are automatically inserted by the scheduler, and are invisible to the user. Moreover, these stars can not be scheduled. They are added to sub-galaxies only for the allocation of memory and other resources before generating code. The "spread" and "collect" stars themselves do not require extra memory since in most cases we can overlay memory buffers. For example, in the first example, a buffer of size 2 is assigned to the output of star A. Star B obtains the information it needs to fetch a token from the first location of the buffer via the "spread" star, while the "send" star knows that it will fetch a token from the second location. Thus, the buffers for the outputs of the "spread" star are overlaid with the output buffer of star A.

In case there are delays or past tokens are used on the connection between two blocks that should be connected through "spread" or "collect" stars, we need to copy data explicitly. Thus, we will need extra memory for these stars. In this case, the user will see the existence of "spread/collect" stars in the generated code.

`Spread/Collect` stars have only been implemented in the CGC domain so far.

12.5 Interface Issues

For the 0.6 release, we have developed a framework for interfacing code generation targets with other targets (simulation or code generation). In this section we will detail how to support this new framework for a code generation target. To learn how to develop applications within Ptolemy that use multiple targets that support this new framework, refer to the *Interface Issues* section in the *User's Manual - CG Domain* chapter.

As with `Wormholes`, we have developed a way to interface N targets without requiring N^2 specialized interfaces. We do this by generating a customized interface (analogous to the universal `EventHorizon` in wormholes) that is automatically built by using communication stars supplied by each code generation target. This interface is generated in C (using the CGC domain) and runs on the Ptolemy host workstation.

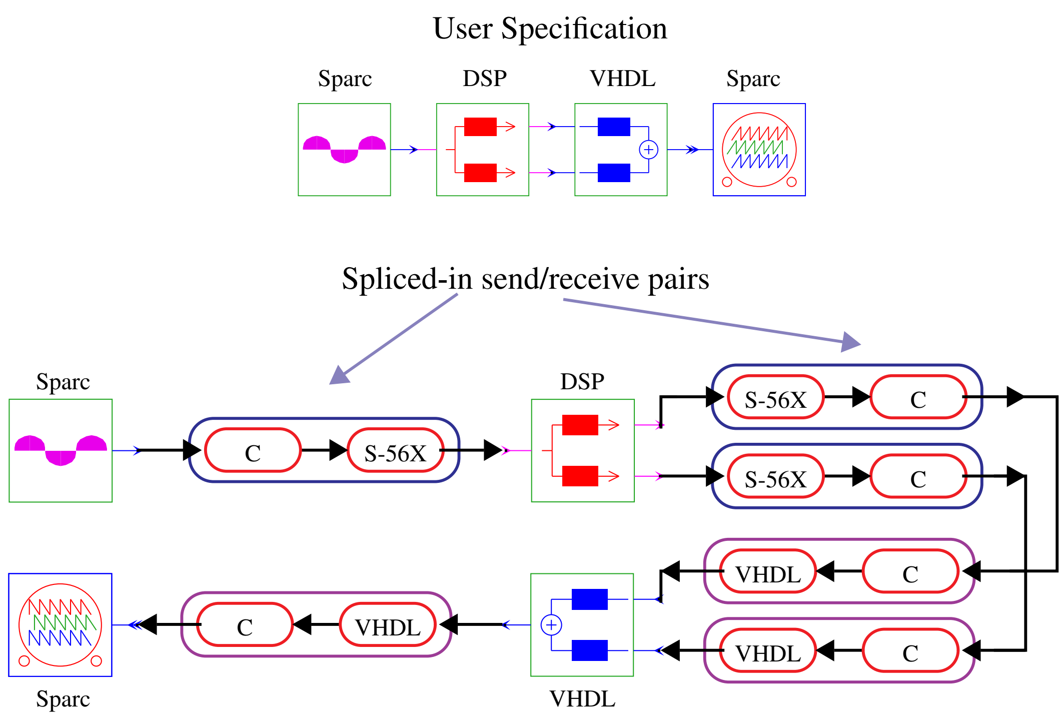

FIGURE 12-1: An interface constructed between three code generation domains. The interface constructed by the framework is made up of communication pairs, each pair encircled by an ellipse. The first (sine) and last (xgraph) stars are to be run on the host workstation (CGC). The second block (analysis filter bank, a galaxy made up of two polyphase FIR actors) is to be run on a DSP card (CG56). The third block (synthesis filter bank, a galaxy made up of two polyphase FIR actors) is to be run using a VHDL simulator.

To support this infrastructure, a target writer needs to define two pairs of communication stars and add target methods which return each of these pairs. The framework will then build the interface by splicing in these stars as is shown in figure 12-1. These same actors are used when constructing an interface to a Ptolemy simulation target as shown in figure 12-2.

These communication stars, described in section 12.4.2, are a specialized form of send/receive stars. In addition to the previous assumptions in section 12.4.2, send/receive for this infrastructure must also define C code to control the target for operations such as downloading, initializing and (if applicable) terminating the generated executable.

One pair of communication stars must communicate from the target to the CGC code that will run on the Ptolemy host workstation. The other pair must communicate in the opposite direction. The CGC send/receive stars are typically defined from a common base communication star specific for each target. This common base defines the C code to control a target that was discussed in the previous paragraphs. Examples of send/receive stars that support this infrastructure can be found in:

For the S56XTarget (Ariel S-56X DSP card):

```
$PTOLEMY/src/domains/cg56/targets/CGCXBase.pl
$PTOLEMY/src/domains/cg56/targets/CGCXSend.pl
$PTOLEMY/src/domains/cg56/targets/CGCXReceive.pl
```

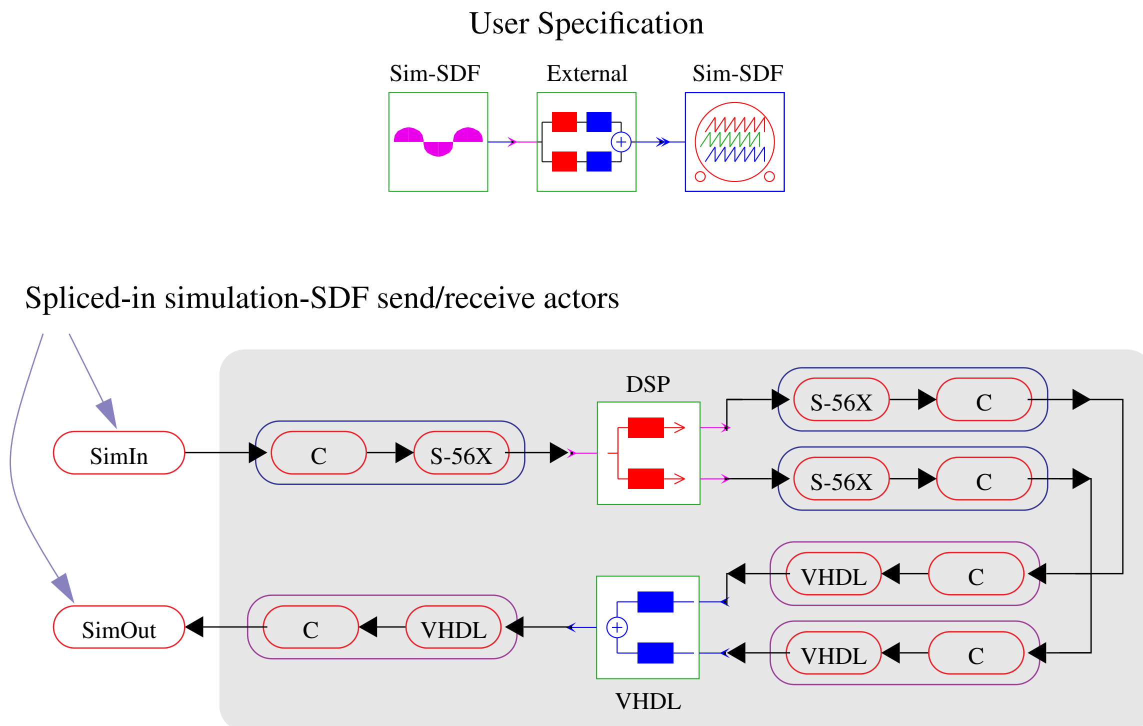

FIGURE 12-2: General Ptolemy simulation interface. The analysis and synthesis filter bank blocks are identical to those described in figure 12-1. The SimIn and SimOut stars are built into Ptolemy and defined in:
$PTOLEMY/src/domains/cgc/targets/main/CGCSDF{Send,Receive}.pl

```
$PTOLEMY/src/domains/cg56/targets/CG56XCSend.pl
$PTOLEMY/src/domains/cg56/targets/CG56XCReceive.pl
```

For the SimVSSTarget (Synopsis VSS Simulator):

```
$PTOLEMY/src/domains/vhdl/targets/CGCVSynchComm.pl
$PTOLEMY/src/domains/vhdl/targets/CGCVSend.pl
$PTOLEMY/src/domains/vhdl/targets/CGCVReceive.pl
$PTOLEMY/src/domains/vhdl/targets/VHDLCSend.pl
$PTOLEMY/src/domains/vhdl/targets/VHDLCReceive.pl
```

After defining both pairs of communication stars, methods to instantiate these stars must be defined in the target:

```
CommPair fromCGC(PortHole&);
CommPair toCGC(PortHole&);
```

A `CommPair` is a communication pair, where one of the communication stars in a CGC star. The `S56XTarget::fromCGC` method, illustrates the typical code needed for these methods:

```
CommPair S56XTarget::fromCGC(PortHole&) {
      CommPair pair(new CGCXSend,new CG56XCReceive);
      configureCommPair(pair);
      return pair;
}
```

The `configureCommPair` function is defined in the `S56XTarget.cc` file and configures the S56XTarget communication stars.