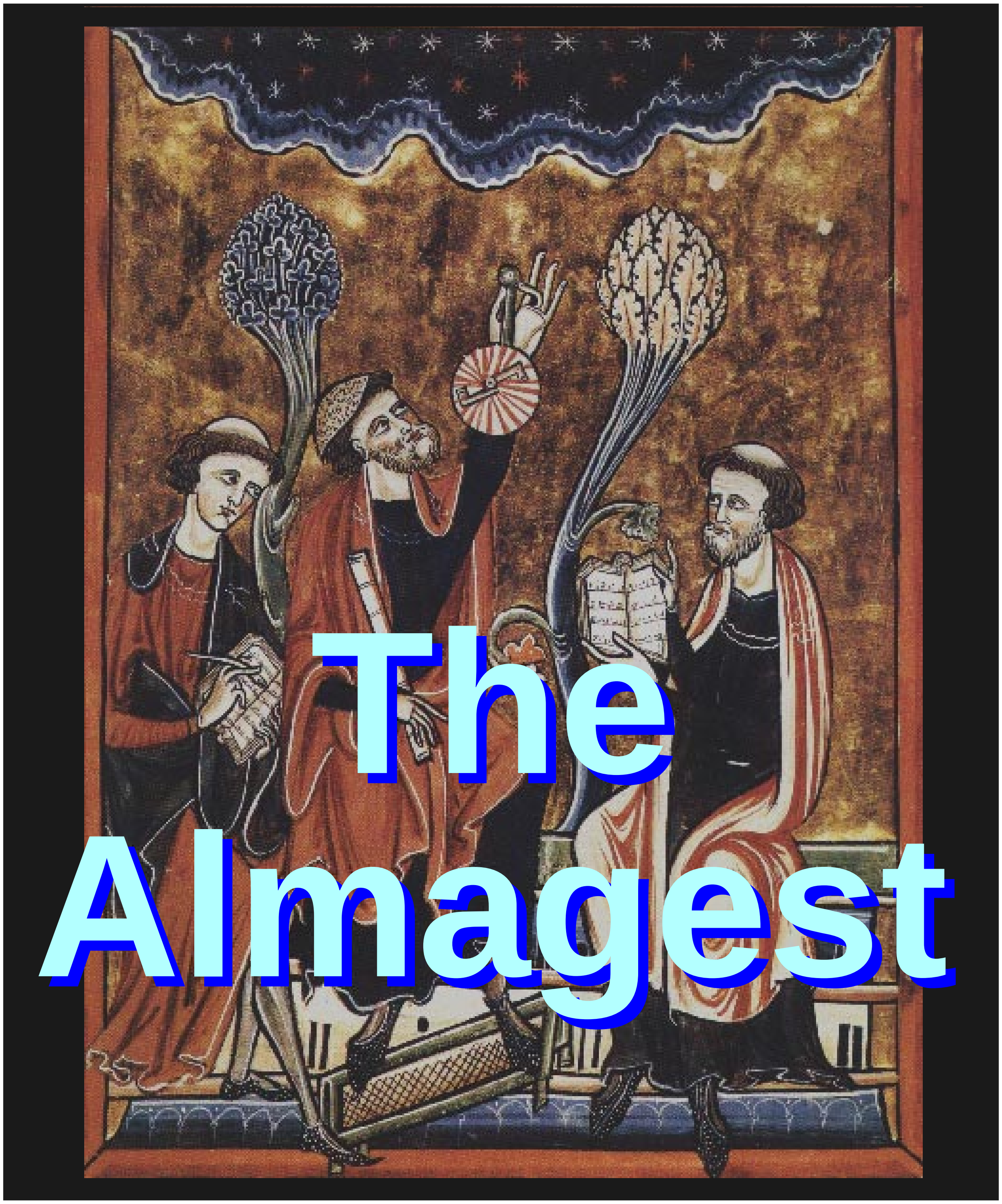
UNIVERSITY OF CALIFORNIA AT BERKELEY

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



The Almagest

Vol. 3 - Ptolemy 0.7 Kernel Manual

Primary Authors

Joseph T. Buck (Part I — The Kernel Manual, Chapters 1-12)

Soonhoi Ha (Part II — Code Generation, Chapters 13-18)

Other contributors

Shuvra Bhattacharyya, Wan-Teh Chang, Michael J. Chen, John S. Davis II, Brian L. Evans, Mudit Goel,Christopher Hylands, Asawaree Kalavade, Alan Kamas, Tom Lane, Bilung Lee, Edward A. Lee, Jie Liu, David G. Messerschmitt, Praveen Murthy, Thomas M. Parks, José Luis Pino, Gilbert Sih, Neil Smyth, S. Sriram, Michael C. Williamson, Kennard White,Yuhong Xiong

Current Sponsors

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

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

Trademarks

About the Cover

The image on the cover is from a medieval illuminated manuscript at the Bibliotheque Nationale, Paris. It depicts a monk using an *astrolabe*, a device for measuring the angular distance of stars above the horizon. An assistant records the readings.

Contents

Introduction

The Ptolemy Kernel Manual describes the C++ classes that make up the core of Ptolemy. It is assumed that the reader is intimately familiar with the Ptolemy system (If you are not, see `http://ptolemy.eecs.berkeley.edu`). The Kernel Manual was originally written by Joe Buck, Soonhoi Ha added the Code Generation chapters. The last version of the Kernel Manual was released with Ptolemy 0.5.2. With the release of Ptolemy 0.7, we decided that it was time to update the Kernel Manual to include things like Tom Lane's changes to the type checking system ("The PortHole type resolution algorithm" on page 6-15). However, much of the Kernel Manual is still out of date, the most common errors are changes in the calling signature of methods, or the addition or removal of methods. Ultimately, the source code is the best reference for these sorts of issues. We decided to release the Kernel Manual in an unpolished form because the usefulness of some of the sections far outweighs the bugs in other sections.

Chapter 1. Basic concepts, classes, and facilities

Authors: *The Ptolemy Team*

This section describes some basic classes and low-level concepts that are used throughout Ptolemy. There are a number of iterator classes, all with the same interface. Several important non-class library functions are provided. A basic linked list class called SequentialList is heavily used. States (see section 9.1) and Portholes (see section 6.2) can have *attributes*; these are particularly important in code generation. Finally, many of the significant classes in Ptolemy – functional blocks, portholes to implement connections, parameters – are derived from NamedObj, the basic object for implementing a named object that lives in a hierarchy.

1.1 The C++ Subset Used In Ptolemy

The Ptolemy system has grown up with the C++ language, so it does not use all the latest features in the newest compilers or every nook and cranny of Ellis and Stroustrup's Annotated Reference Manual, because of unimplemented features or lack of stability of implementation. Instead, we focused on stability. Accordingly, Ptolemy can be built with a number of different C++ compilers. This means, for one thing, that templates are not used (except in the experimental IPUS domain). In addition, some features that do not work that well yet under g++, such as nested classes, are also avoided. Nested enumerations, however, are used in several places.

1.2 Iterators

Iterators are a very basic and widely used concept in Ptolemy, and are used repeatedly in Ptolemy programming in almost any situation where a composite object contains other objects. We have chosen to use a consistent interface for all iterator objects. The typical iterator class has the following basic interface (some iterators provide additional functions as well):

```
class MyIterator {
public:
// constructor: argument is associated outer object
      MyIterator(OuterObject&);
      // next: return a pointer to the next object,
      // or a null pointer if no more
      InnerObject* next();
      // operator form: a synonym for next
      InnerObject* operator++(POSTFIX_OP) {return next();}
      // reset the iterator to point to the first object
      void reset();
}
```

`POSTFIX_OP` is a macro that is defined to be an empty string on older compilers (such as cfront 2.1 and versions of g++ before 2.4) and to the string "int" with newer compilers. This conditional behavior is required because of the evolution of the C++ language; previously, postfix and prefix forms of the operators ++ and – were not distinguished when overloaded; now, a dummy int argument indicates that the postfix form is intended.

A typical programming application for iterators might be something like

```
// print the names of all objects in the container
ListIter nextItem(myList);
Item *itemP;
while ((itemP = nextItem++) != 0)
      cout << itemP->name() << "\back n";
```

It is, as a rule, not safe to modify most container classes in parallel with the use of an iterator, as the iterator may attempt to access an object that does not exist any more. However, the `reset` member function will always make the iterator safe to use even if the list has been modified (user-written iterators should preserve this property).

1.3 Non-class utility procedures

The kernel provides several useful ordinary (non-class) procedures, primarily for manipulating strings and path names. Some are defined in `miscFuncs.h,` others in `paths.h.`

`char* savestring(const char*` *`text`*`);`

Create a copy of the *`text`* argument with `new` and return a pointer to it. It is the caller's responsibility to assure that the string is eventually deleted by using the `delete` [] operator. The argument *`text`* must not be a null pointer.

`const char* hashstring(const char*` *`text`*`);`

Enters a copy of *`text`* into a hash table and return a pointer to the entry. If two strings compare equal when passed to `strcmp,` then if both are passed to `hashstring,` the return values will be the same pointer.

`const char* expandPathName(const char*` *`fileName`*`);`

Expand a path name that may begin with an environment variable or a user's home directory. If the string does not begin with a ~ or `$` character, the string itself is returned. A leading "`~/`" is replaced by the user's home directory; a leading "~*`user`* " is replaced by the home directory for *`user,`* unless there is no such user, in which case the original string is returned. Finally, a leading "`$`*`env`*" is replaced by the value of the environment variable *`env;`* if there is no such environment variable, the original string is returned. Note that references to environment variables other than at the beginning are *not* substituted. If any substitutions are made, the return value is actually a pointer into a static buffer. This means that a second call to this function may write on top of a value returned by a previous call.

`const char* pathSearch(const char*` *`file`*`, const char*` *`path`*`=0);`

For this function, *`path`* is a series of Unix-style directory names, separated by colons. If no second argument is supplied or if the value is null, the value of the PATH environment variable is used instead. For each of the colon-separated directory strings, the function checks to see whether the file exists in the named directory. If it finds a match, it returns a pointer to an internal buffer containing the full path of the match. If it does not find a match, it returns a null pointer.

`int progNotFound(const char* `*`program`*`,const char* `*`extra`*`=0);`

This function searches for *`program`* in the user's PATH using the `pathSearch` function. If a match is found, the function returns false (0). Otherwise it returns true (1) and also generates an error message with the `Error::abortRun` function. If the *`extra`* argument is given, it forms the second line of the error message.

1.4 Generic Data Structures

As Ptolemy does not use templates, our generic lists use the generic pointer technique, with

```
typedef void * Pointer;
```

The most commonly used generic data structure in Ptolemy is `SequentialList.` Other lists are, as a rule, privately inherited from this class, so that type safety can be preserved. It is possible to insert and retrieve items at either the head or the tail of the list.

1.5 Class SequentialList

This class implements a single linked list with a count of the number of elements. The constructor produces a properly initialized empty list, and the destructor deletes the links. However, the destructor does not delete the items that have been added to the list; this is not possible because it has only `void` * pointers and would not know how to delete the items. There is an associated iterator class for SequentialList called ListIter.

1.5.1 SequentialList information functions

These functions return information about the SequentialList but do not modify it.

`int size() const;`

Return the size of the list.

`Pointer head() const;`

Return the first item from the list (0 if the list is empty). The list is not changed.

`Pointer tail() const;`

Return the last item from the list (0 if the list is empty). The list is not changed.

`Pointer elem(int `*`n`*`) const;`

Return the *`nth`* item on the list (0 if there are fewer than *`n`* items). Note that the time required is proportional to *`n.`*

`int empty() const;`

Return 1 if the list is empty, 0 if it is not.

`int member(Pointer `*`arg`*`) const;`

Return 1 if the list has a Pointer that is equal to *`arg,`* 0 if not.

1.5.2 Functions that modify a SequentialList

`void prepend(Pointer `*`p`*`);`

Add an item at the beginning of the list.

`void append(Pointer p);`

Add an item at the end of the list.

`int remove(Pointer p);`

Remove the pointer *p* from the list if it is present (the test is pointer equality). Return 1 if present, 0 if not.

`Pointer getAndRemove();`

Return and remove the head of the list. If the list is empty, return a null pointer (0).

`Pointer getTailAndRemove();`

Return and remove the last item on the list.

`void initialize();`

Remove all links from the list. This does not delete the items pointed to by the pointers that were on the list.

1.5.3 Class ListIter

ListIter is a standard iterator class for use with objects of class SequentialList. The constructor takes an argument of type `const SequentialList` and the ++ operator (or `next` function) returns a `Pointer.` Class ListIter is a friend of class SequentialList. In addition to the standard iterator functions `next` and `reset,` this class also provides a function

```
void reconnect(const SequentialList& newList)
```

that attaches the ListIter to a different SequentialList.

1.6 Doubly linked lists

Support for doubly linked lists is found in `DoubleLink.h.` The class DoubleLink implements a base class for nodes in the list, class DoubleLinkList implements the list itself, and class DoubleLinkIter forms an iterator. *WARNING: We consider this class to have serious design flaws, so it may be reworked quite a bit in subsequent Ptolemy releases*.

1.6.1 Class DoubleLink

A DoubleLink object is an item in the list defined by DoubleLinkList. Normally, a programmer will not interact directly with this class, but rather will use methods in DoubleLinkList. Nonetheless, we present it here because some of the methods of DoubleLinkList do refer to it.

There are two constructors:

`DoubleLink(Pointer p, DoubleLink* next, DoubleLink* prev):`
`DoubleLink(Pointer p);`

The first form initializes the *next* and *prev* pointers of the node as well as the contents. The second form sets these pointers to null and only initializes the contents pointer.

`Pointer content();`

Return the content pointer of the node.

`virtual ~DoubleLink();`
: The destructor is virtual.

`void unlinkMe();`
: Delete the node from the list it is contained in. I.e. connect the elements pointed to by the *`prev`* and *`next`* pointers. The object pointed to by the node is not deleted.

The following data members are protected:

```
DoubleLink *next; // next node in the list
DoubleLink *prev; // previous node in the list
Pointer e;        // contents of this node
```

1.6.2 Class DoubleLinkList

`DoubleLinkList();`
`DoubleLinkList(Pointer* e);`
: The first constructor creates an empty list. The second creates a one-node list containing the object pointed to by *`e`*. That object must live at least as long as the link lives.

`virtual ~DoubleLinkList();`
: The destructor is virtual. It deletes all DoubleLinks in the list, but does not delete the objects pointed to by each link.

`DoubleLink* createLink(Pointer e);`
: Return a newly allocated DoubleLink that contains a pointer to *`e`*. It is up to the caller to delete the DoubleLink, or to use either `removeLink` or `remove`.

`void insertLink(DoubleLink *x);`
`void insert(Pointer e);`
: These methods insert an item at the beginning of the list. The first inserts a DoubleLink; the second creates a DoubleLink with `createLink` and inserts that. If the second form is used, the link should only be removed using `removeLink` or `remove`, not `unlink`, because `unlink` will not delete the DoubleLink.

`void appendLink(DoubleLink *x);`
`void append(Pointer e);`
: These methods append at the end of the list. The first appends a DoubleLink; the second creates a DoubleLink with `createLink` and appends that. If the second form is used, the link should only be removed using `removeLink` or `remove`, not `unlink`, because `unlink` will not delete the DoubleLink.

`void insertAhead(DoubleLink *y, DoubleLink *x);`
`void insertBehind(DoubleLink *y, DoubleLink *x);`
: The first method inserts *`y`* immediately ahead of the DoubleLink pointed to by*`x`*; the second inserts *`y`* immediately after the DoubleLink pointed to by*`x`*. Both of these functions assume that `x` is in the list; disaster may result otherwise.

`DoubleLink* unlink(DoubleLink *x);`
: Remove the link *`x`* from the list and return a pointer to it. Make sure that *`x`* is in the list

before calling this method, or disaster may result.

`void removeLink(DoubleLink *x);`

This is the same as `unlink,` except that *x* is deleted as well. The same cautions apply.

`void remove(Pointer e);`

Search for a DoubleLink whose contents match *e*. If a match is found, the node is removed from the list and the DoubleLink is deleted. The object pointed to by *e* is not deleted. The search starts at the head of the list.

`int find(Pointer e);`

Search for a DoubleLink whose contents match *e*. If a match is found, 1 (true) is returned; otherwise 0 (false) is returned. The search starts at the head of the list.

`virtual void initialize();`

Delete all DoubleLinks in the list and make the list empty.

`void reset();`

Make the list empty, but do not delete the DoubleLinks in each of the nodes.

`int size();`

Return the number of elements in the list. This method should be const but isn't.

```
DoubleLink *head();
DoubleLink *tail();
```

Return a pointer to the head or to the tail of the list. If the list is empty both methods will return a null pointer.

```
DoubleLink *getHeadLink();
Pointer takeFromFront();
```

The first method gets and removes the head link, returning a pointer to it. The second method returns the object pointed to by the head link, and deletes the DoubleLink. If the list is empty, both functions return a null pointer.

```
DoubleLink *getTailLink();
Pointer takeFromBack();
```

These methods are identical to the previous pair except that they remove the last node rather than the first.

The following two data members are protected:

```
DoubleLink *myHead;
DoubleLink *myTail;
```

1.6.3 Class DoubleLinkIter

DoubleLinkIter is an iterator for DoubleLinkList. It is only capable of moving "forward" through the list (following the "next" links, not the "prev" links). Its `next` operator returns the

Pointer values contained within the nodes; it is also possible to use the non-standard `nextLink` function to return successive DoubleLink pointers.

1.7 Other generic container classes

The file `DataStruct.h` defines two other generic container classes that are privately derived from SequentialList: Queue and Stack. Class Queue may be used to implement a FIFO or a LIFO queue, or a mixture. Class Stack implements a stack.

1.7.1 Class Queue

The constructor for class Queue builds an empty queue. The following four functions move pointers into or out of the queue:

```
void putTail(Pointer p);
void putHead(Pointer p);
Pointer getHead();
Pointer getTail();
```

In addition, `put` is a synonym for `putTail,` and `get` is a synonym for `getHead.` All these functions are implemented on top of the (hidden) SequentialList functions. The SequentialList functions `size` and `initialize` are re-exported (that is, are accessible as public member functions of class Stack).

1.7.2 Class Stack

The constructor for class Stack builds an empty stack. The following functions move pointers onto or off of the stack:

```
void pushTop(Pointer p);
Pointer popTop();
pushBottom(Pointer p);
```

`pushTop` and `popTop` are the functions traditionally associated with a stack; `pushBottom` adds an item at the bottom, which is non-traditional. The following non-destructive function also exists:

```
Pointer accessTop() const;
```

This accesses but does not remove the element from the top of the stack.

All these functions are implemented on top of the (hidden) SequentialList functions. The SequentialList functions `size` and `initialize` are re-exported.

1.8 Class NamedObj

NamedObj is the base class for most of the common Ptolemy objects. A NamedObj is, simply put, a named object; in addition to a name, a NamedObj has a pointer to a parent object, which is always a Block (a type of NamedObj). This pointer can be null. A NamedObj also has a descriptor. Warning! NamedObj assumes that the name and descriptor “live” as long as the NamedObj does. They are not deleted by the destructor, so that they can be compile-time strings. Important derived types of NamedObj include Block (see section3.1), GenericPort (see section 6.1), State (see section 9.1), and Geodesic (see section 6.6).

1.8.1 NamedObj constructors and destructors

All constructors and destructors are public. NamedObj has a default constructor, which sets the name and descriptor to empty strings and the parent pointer to null, and a three-argument constructor:

`NamedObj(const char*` *`name`*`,Block*` *`parent`*`, const char*` *`descriptor`*`)`

NamedObj's destructor is virtual and does nothing.

1.8.2 NamedObj public members

`virtual const char* className() const;`

Return the name of the class. This needs to have a new implementation supplied for every derived class (except for abstract classes, where this is not necessary).

`const char* name() const;`

Return the local portion of the name of the class (vs. the full name).

`const char* descriptor() const;`

Return the descriptor.

`Block* parent() const;`

Return a pointer to the parent block.

`virtual StringList fullName() const;`

Return the full name of the object. This has no relation to the class name; it specifies the specific instance's place in the universe-galaxy-star hierarchy. The default implementation returns names that might look like `universe.galaxy.star.port` for a porthole; this is the full name of the parent, with a period and the name of the object appended.

`void setName(const char*` *`name`*`);`

Set the name of the object. The string must live at least as long as the object.

`void setParent(Block*` *`parent`*`);`

Set the parent of the object, which is always a Block. The parent must live at least as long as the object.

`void setNameParent (const char*` *`my_name`*`, Block*` *`my_parent`*`)`

Change the name and parent pointer of the object.

`virtual void initialize() = 0;`

Initialize the object to prepare for system execution. This is a pure virtual method.

`virtual StringList print (int` *`verbose`*`) const;`

Return a description of the object. If the argument *`verbose`* is 0, a somewhat more compact form is printed than if the argument is non-zero.

`virtual int isA(const char*` *`cname`*`) const;`

Return TRUE if the argument is the name of the class or of one of its base classes. This method needs to be redefined for all classed derived from NamedObj. To make this easy to do, a macro `ISA_FUNC` is provided; for example, in the file `Block.cc` we see the line

```
ISA_FUNC(Block,NamedObj);
```

NamedObj is the base class from which Block is derived. This macro creates the function definition

```
int Block::isA(const char* cname) const {
if (strcmp(cname,"Block") == 0) return TRUE;
      else return NamedObj::isA(cname);
}
```

Methods `isA` and `className` are overridden in all derived classes; the redefinitions will not be described for each individual class.

1.8.3 Flags on named objects

`FlagArray flags`

Many schedulers and targets need to be able to mark blocks in various ways, to count invocations, or flag that the block has been visited, or to classify it as a particular type of block. To support this, we provide an array of flags that are not used by class Block, and may be used in any way by a Target. The target may defer their use to its schedulers. The array can be of any size, and the size will be increased automatically as elements are referenced. For readability and consistency, the user should define an enum in the target or scheduler class to give the indices, so that mnemonic names can be associated with flags, and so that multiple schedulers for the same target are consistent. For instance, if `b` is a pointer to a `Block,` a target might contain the following:

```
private:
      enum {
            visited = 0,
            fired = 1
            }
```

which can then be used in code to set and read flags in a readable way,

```
b->flags[visited] = TRUE;
   ...
if (b->flags[visited]) { ... }
```

WARNING: For efficiency, there is no checking to prevent two different pieces of code (say a target and scheduler) from using the same flags (which are indexed only by non-negative integers) for different purposes. The policy, therefore, is that *the target is in charge*. It is incumbent upon the writer of the target to know what flags are used by schedulers invoked by that target, and to avoid corrupting those flags if the scheduler needs them preserved. We weighed a more modular, more robust solution, but ruled in out in favor of something very lightweight and fast.

1.8.4 NamedObj protected members

`void setDescriptor(const char* desc);`

Set the descriptor to *`desc.`* The string pointed to by *`desc`* must live as long as the NamedObj does.

1.9 Class NamedObjList

Class NamedObjList is simply a list of objects of class NamedObj. It is privately inherited from class SequentialList (see section 1.5), and, as a rule, other classes privately inherit from it. It supports only a subset of the operations provided by SequentialList; in particular, objects are added only to the end of the list. It provides extra operations, like searching for an object by name and deleting objects. This object enforces the rule that only const pointers to members can be obtained if the list is itself const; hence, two versions of some functions are provided.

1.9.1 NamedObjList information functions

The `size` and `initialize` functions of SequentialList are re-exported. Note that `initialize` removes only the links to the objects and does not delete the objects. Here is what's new:

```
NamedObj* objWithName(const char* name);
const NamedObj* objWithName(const char* name) const;
```

Find the first NamedObj on the list whose name is equal to *`name,`* and return a pointer to it. Return 0 if it is not found. There are two forms, one of which returns a const object.

```
NamedObj* head();
const NamedObj* head() const;
```

Return a pointer to the first object on the list (0 if none). There are two forms, one of which returns a const object.

1.9.2 Other NamedObjList functions

`void put(NamedObj& obj)`

Add a pointer to *`obj`* to the list, at the end. The object must live at least as long as the list.

`void initElements();`

Apply the `initialize` method to each NamedObj on the list.

`int remove(NamedObj* obj);`

Remove *`obj`* from the list, if present (this does not delete *`obj).`* Return 1 if it was present, 0 if not.

`void deleteAll();`

Delete all elements from the list, and reset it to be an empty list. WARNING: this assumes that the members of the list are on the heap (allocated by `new,` so that deleting them is valid)!

1.9.3 NamedObjList iterators

There are two different iterators associated with NamedObjList; class NamedObjListIter and class CNamedObjListIter. The latter returns const objects (which cannot then be modified). The former returns a non-const pointer, and can only be used if the NamedObjList itself is not const. Both obey the standard iterator interface and are privately derived from class ListIter.

1.10 Attributes

Attributes represent logical properties that an object may or may not have. Certain classes such as State and Porthole contain attributes and provide interfaces for setting and clearing attributes. For the State class, for instance, the initial value may or may not be settable by the user; this is indicated by an Attribute. In code generation classes, attributes may indicate whether an assembly-language buffer should be allocated to ROM or RAM, fast memory or slow memory, etc. The set of attributes of an object is stored in an entity called a `bitWord`. At present, a bitWord is represented as an unsigned long, which restricts the number of distinct attributes to 32; this may be changed in future releases. An Attribute object represents a request to turn certain attributes of an object off, and to turn other attributes on. As a rule, constants of class Attribute are used to represent attributes, and users have no need to know whether a given property is represented by a true or false bit in the bitWord. Although we would prefer to have a constructor for Attribute objects of the form

```
Attribute(bitWord bitsOn, bitWord bitsOff);
```

it has turned out that doing so presents severe problems with order of construction, since a number of global Attribute objects are used and there is no simple, portable way of guaranteeing that these objects are constructed before any use. As a result, the `bitsOn` and `bitsOff` members are public, but we forbid use of that fact except in one place: constant Attribute objects can be initialized by the C “aggregate form”, as in the following example:

```
extern const Attribute P_HIDDEN = {PB_HIDDEN, 0};
```

(This particular attribute is used by Porthole to indicate that a port should not be visible to the user, i.e. should not appear on an icon.) The first word specified is the `bitsOn` field, `PB_HIDDEN`, and the second word specified is the `bitsOff` field. Other than to initialize objects, we pretend that these data members are private.

1.10.1 Attribute member functions

```
Attribute& operator |= (const Attribute& arg);
Attribute& operator &= (const Attribute& arg);
```

These operations combine attributes, by applying the `|=` and `&=` operators to the bitsOn and bitsOff fields. The first operation, as attributes are commonly used, represents a requirement that two sets of attributes be met, so it has been argued that it really should be the “and” operation. However, the current scheme has the virtue of consistency.

```
bitWord eval(bitWord defaultVal) const;
```

Evaluate an attribute given a default value. Essentially, bits corresponding to bitsOn are turned on, and then bits corresponding to bitsOff are turned off.

`bitWord clearAttribs(bitWord` *`defaultVal`*`) const;`

This method essentially applies the attribute backwards, reversing the roles of bitsOn and bitsOff in `eval`.

```
bitWord on() const;
bitWord off() const;
```

Retrieve the bitsOn and bitsOff values, respectively. Inline definitions of operators `&` and `|` are also defined to implement nondestructive forms of the `&=` and `|=` operations.

1.11 FlagArray

`FlagArray` is a lightweight, self-expanding array of integers. It is meant to store an array of flags or counters, and its main appearance in Ptolemy is as a public member of class `NamedObj`, and therefore is available in most Ptolemy classes, which are derived from `NamedObj`. Targets and schedulers use this member to keep track of various kinds of data. Many schedulers and targets need to be able to mark blocks in various ways, for example to count invocations, or flag that the block has been visited, or to classify it as a particular type of block. This class provides a simple mechanism for doing this. A `FlagArray` object is indexed like an array, using square brackets. If `x` is a `FlagArray` and `i` is a non-negative integer, then `x[i]` is a reference to an integer element of the array. If `i` is out of bounds (beyond the currently allocated limits of the array), then the class automatically increases the size of the array. New elements are filled with zeros. Thus, a `FlagArray` may be viewed as an infinite dimensional array of integers initialized with zeros. If `i` is a negative integer, then `x[i]` is an error. For efficiency, the class does not test for this error at run time, so you could get a core dump if you make this error.

1.11.1 FlagArray constructors and destructor

`FlagArray()`

This constructor creates a zero-length flag array.

`FlagArray(int size)`

This constructor creates a flag array with the specified size already allocated and filled with zeros.

`FlagArray(int` *`size`*`, int` *`fill_value`*`)`

This constructor creates a flag array with the specified size filled with the specified integer value. The destructor frees the memory allocated to store the array of integers.

1.11.2 FlagArray public methods

`FlagArray & operator = (const FlagArray &` *`v`*`)`

An assignment to one `FlagArray` from another simply copies its size and data.

`int size() const`

Return the current allocated size of the array.

`int & operator [] (int` *`n`*`)`

If *n* is less than the currently allocated size of the array, then this returns a reference to the n-th element of the array. If *n* is greater than or equal to the currently allocated size of the array, then the size of the array is increased, the new elements are filled with zeros, and a reference to the n-th element is returned. Indexing of elements begins with zero. The returned reference, of course, can be used on the left-hand side of an assignment. This is how values are written into an array.

Chapter 2. Support for multithreading

Authors: *Joseph T. Buck*

Multithreading means that there are multiple *threads of control*, or *lightweight processes*, in the same Ptolemy process. The principal consequence of the existence of multithreading is that it is necessary to provide mechanisms that guarantee exclusive access to resources. The Ptolemy kernel does not provide a multithreading library, as this is currently a very OS and CPU-specific operation. There are a variety of such libraries that might be used; Sun's lightweight processes library and the University of Colorado's Awesime package are two examples. What the kernel does provide is a locking mechanism for implementing *critical regions*, noninterruptable regions of code in which only one thread can be active at a time. This facility is used to protect critical resources in the kernel that might be accessed by multiple threads.

2.1 Class PtGate

Objects of classes derived from PtGate are used as semaphores to obtain exclusive access to some resource. PtGate is an abstract base class: it specifies certain functionality but does not provide an implementation. Derived classes typically provide the desired semantics for use with a particular threading library. PtGate has three virtual functions that must be implemented by each derived class. The first is a public method:

```
virtual PtGate* makeNew() const = 0;
```

The `makeNew` method returns a new object of the same class as the object it is called for, which is created on the heap. For example, a hypothetical SunLWPGate object would return a new SunLWPGate. The other two methods are protected. They are:

```
virtual void lock() = 0;
virtual void unlock() = 0;
```

The first call requests access for a resource; the second call releases access. If code in another thread calls `lock()` on the same PtGate after `lock()` has already been called on it, the second call will block until the first thread does an `unlock()` call. Note that two successive calls to `lock()` on the same PtGate from the same thread will cause that thread to hang. It is for this reason that these calls are protected, not public. Access to PtGates by user code is accomplished by means of another class, CriticalSection. The CriticalSection class is a friend of class PtGate.

2.2 Class CriticalSection

CriticalSection objects exploit the properties of constructors and destructors in C++ to provide a convenient way to implement *critical sections*: sections of code whose execution can be guaranteed to be atomic. Their use ensures that data structures can be kept consistent even when accessed from multiple threads. The CriticalSection class implements no methods other than constructors and a destructor. There are three constructors:

```
CriticalSection(PtGate *);
CriticalSection(PtGate &);
CriticalSection(GateKeeper &);
```

The function of all of these constructors is to optionally set a lock. The first constructor will set the lock on the given PtGate unless it gets a null pointer; the second form always sets the lock. The third form takes a reference to an object known as a GateKeeper (discussed in the next section) that, in a sense, may "contain" a PtGate. If it contains a PtGate, a lock is set; otherwise no lock is set. The lock is set by calling `lock()` on the PtGate object. The CriticalSection destructor frees the lock by calling `unlock()` on the PtGate object, if a lock was set. CriticalSection objects are used only for their side effects. For example:

```
PtGate& MyClass::gate;
...
void MyClass::updateDataStructure() {
        CriticalSection region(MyClass::gate);
        code;
        ...
}
```

The code between the declaration of the CriticalSection and the end of its scope will not be interrupted.

2.3 Class GateKeeper

The GateKeeper class provides a means of registering a number of PtGate pointers in a list, together with a way of creating or deleting a series of PtGate objects all at once. The motivation for this is that most Ptolemy applications do not use multithreading, and we do not wish to pay the overhead of locking and unlocking where it is not needed. We also want to have the ability to create a number of fine-grain locks all at once. GateKeeper objects should be declared only at file scope (never as automatic variables or on the heap). The constructor takes the form

```
GateKeeper(PtGate *& gateToKeep);
```

The argument is a reference to a pointer to a GateKeeper, and the function of the constructor is to add this reference to a master list. It will later be possible to "enable" the pointer, by setting it to point to a newly created PtGate of the appropriate type, or "disable" it, by deleting the PtGate object and setting the pointer to null. The GateKeeper destructor deletes the reference from the master list and also deletes any PtGate object that may be pointed to by the PtGate pointer. The public method

```
int enabled() const;
```

returns 1 if the GateKeeper's PtGate pointer is enabled (points to a PtGate) and 0 otherwise (the pointer is null). There are two public static functions:

`static void enableAll(const PtGate&` *`master`*`);`

This function creates a PtGate object for each GateKeeper on the list, by calling `make-New()` on the master object.

```
static void disableAll();
```

This function destroys all the PtGate objects and sets the pointers to be null. This function must never be called from within a block controlled by a CriticalSection, or while multithreading operation is in progress. A GateKeeper may be used as the argument to a CriticalSection constructor call; the effect is the same as if the PtGate pointer were passed to the constructor directly.

2.4 Class KeptGate

A KeptGate object is simply a GateKeeper that contains its own PtGate pointer. It is derived from GateKeeper, has a private PtGate pointer member, and a constructor with no arguments. Like a GateKeeper, it should be declared only at file scope and may be used as an argument to a CriticalSection constructor call.

Chapter 3. Block and related classes

Authors: *Joseph T. Buck*

Other Contributors: *J. Liu*

This section describes Block, the basic functional block class, and those objects derived from it. It is Blocks more than anything else that a user of Ptolemy deals with. Actors as well as collections of actors are Blocks. Although the Target class is derived from class Block, it is documented elsewhere, as it falls under control of execution .

3.1 Class Block

Block is the basic object for representing an actor in Ptolemy. It is derived from NamedObj (see section 1.8). Important derived types of Block are Star (see section 3.2), representing an atomic actor; Galaxy , representing a collection of actors that can be thought of as one actor, and Universe , representing an entire runnable system. A Block has portholes (connections to other blocks — , states (parameters and internal states — , and multiportholes (organized collections of portholes — . While the exact data structure used to represent each is a secret of class Block, it is visible that there is an order to each list, in that iterators return the contained states, portholes, and multiportholes in this order. Iterators are a set of helper classes that step through the states, portholes, or multiportholes that belong to the Block, see the menu entry. Furthermore, Blocks can be cloned, an operation that produces a duplicate block. There are two cloning functions: `makeNew,` which resembles making a new block of the same class, and `clone,` which makes a more exact duplicate (with the same values for states, for example). This feature is used by the KnownBlock class to create blocks on demand.

3.1.1 Block constructors and destructors

Block has a default constructor, which sets the name and descriptor to empty strings and the parent pointer to null, and a three-argument constructor:

`Block(const char* `*`name`*`,Block* `*`parent`*`, const char* `*`descriptor`*`);`

Block's destructor is virtual and does nothing, except for the standard action of destroying the Block's data members. In addition, Block possesses two types of "virtual constructors", the public member functions `makeNew` and `clone.`

3.1.2 Block public "information" members

```
int numberPorts() const;
int numberMPHs() const;
int numberStates() const;
```

The above functions return the number of ports, the number of multiports, or the number of states in the Block.

```
virtual int isItAtomic() const;
virtual int isItWormhole() const;
```

These functions return `TRUE` or `FALSE`, based on whether the Block is atomic or not, or a wormhole or not. The base implementations return `TRUE` for isItAtomic, `FALSE` for isItWormhole.

```
virtual StringList print(int verbose) const;
```

Overrides `NamedObj::print.` This function gives a basic printout of the information in the block.

```
GenericPort* genPortWithName(const char* name);
PortHole* portWithName(const char* name);
MultiPortHole* multiPortWithName(const char* name);
virtual State *stateWithName(const char* name);
```

These functions search the appropriate list and return a pointer to the contained object with the matching name. `genPortWithName` searches both the multiport and the regular port lists (multiports first). If a match is found, it returns a pointer to the matching object as a `GenericPort` pointer.

```
int multiPortNames (const char** names, const char** types,
                    int* io, int nMax) const;
```

Get a list of multiport names.

```
StringList printPorts(const char* type, int verbose) const;
```

Print portholes as part of the info-printing method.

```
virtual Scheduler* scheduler() const;
```

Return the controlling scheduler for this block. The default implementation simply recursively calls the `scheduler()` function on the parent, or returns 0 if there is no parent. The intent is that eventually a block with a scheduler will be reached (the top-level universe has a scheduler, and so do wormholes).

```
virtual Star& asStar();
virtual const Star& asStar() const;
```

Return reference to me as a Star, if I am one. Warning: it is a fatal error (the entire program will halt with an error message) if this method is invoked on a Galaxy! Check with `isItAtomic` before calling it.

```
virtual Galaxy& asGalaxy();
virtual const Galaxy& asGalaxy() const;
```

Return reference to me as a Galaxy, if I am one. Warning: it is a fatal error (the entire program will halt) if this method is invoked on a Star! Check with `isItAtomic` before calling it.

```
virtual const char* domain() const;
```

Return my domain (e.g. SDF, DE, etc.)

3.1.3 Other Block public members

```
virtual void initialize();
```

overrides `NamedObj::initialize`. `Block::initialize` initializes the portholes and states belonging to the block, and calls `setup()`, which is intended to be the "user-supplied" initialization function.

`virtual void preinitialize();`

Perform a "pre-initialization" step. The default implementation does nothing. This method is redefined by HOF stars and other stars that need to rewire a galaxy before the main initialization phase starts. Blocks must act safely if preinitialized multiple times (unless they remove themselves from the galaxy when preinitialized, as the HOF stars do). Preinitialize is invoked by `Galaxy::preinitialize`, which see.

`virtual int run();`

This function is intended to "run" the block. The default implementation does nothing.

`virtual void wrapup();`

This function is intended to be run after the completion of execution of a universe, and provides a place for wrapup code. The default does nothing.

`virtual Block& setBlock(const char*` *`name`*`,Block*` *`parent`*`=0);`

Set the name and parent of a block.

`virtual Block* makeNew() const`

This is a very important function. It is intended to be overloaded in such a way that calling it produces a newly constructed object of the same type. The default implementation causes an error. Every derived type should redefine this function. Here is an example implementation of an override for this function:

```
Block* MyClass::makeNew() const { return new MyClass;}
virtual Block* clone() const
```

The distinction between `clone` and `makeNew` is that the former does some extra copying. The default implementation calls `makeNew` and then `copyStates`, and also copies additional members like `flags;` it may be overridden in derived classes to copy more information. The intent is that `clone` should produce an identical object.

`void addPort(PortHole&` *`port`*`)`
`void addPort(MultiPortHole&` *`port`*`)`

Add a porthole, or a multiporthole, to the block's list of known ports or multiports.

`int removePort(PortHole&` *`port`*`)`

Remove *`port`* from the Block's port list, if it is present. `1` is returned if *`port`* was present and `0` is returned if it was not. Note that *`port`* is not deleted. The destructor for class PortHole calls this function on its parent block.

`void addState(State&` *`s`*`);`

Add the state *`s`* to the Block's state list.

`virtual void initState();`

Initialize the States contained in the Block's state list.

`StringList printStates(const char*` *`type`*`,int` *`verbose`*`) const;`

Return a printed representation of the states in the Block. This function is used as part of the Block's `print` method.

`int setState(const char*` *`stateName`*`, const char*` *`expression`*`);`

Search for a state in the block named *`stateName.`* If not found, return `0`. If found, set its initial value to *`expression`* and return `1`.

3.1.4 Block protected members

`virtual void setup();`

User-specified additional initialization. By default, it does nothing. It is called by `Block::initialize` (and should also be called if initialize is redefined).

`Block* copyStates(const Block&` *`src`*`);`

method for copying states during cloning. It is designed for use by clone methods, and it assumes that the src argument has the same state list as the `this` object.

3.1.5 Block iterator classes

There are three types of iterators that may be used on Blocks: BlockPortIter, BlockStateIter, and BlockMPHIter. Each takes one argument for its constructor, a reference to Block. They step through the portholes, states, or multiportholes, of the Block, respectively, using the standard iterator interface. There are also variant versions with a "C" prefix (CBlockPortIter, etc) defined in the file `ConstIters.h` that take a reference to a const Block and return a const pointer.

3.2 Class Star

Class Star represents the basic executable atomic version of Block. It is derived from Block . Stars have an associated Target , an index value, and an indication of whether or not there is internal state. The default constructor sets the target pointer to `NULL`, sets the internal state flag to `TRUE`, and sets the index value to `-1`.

3.2.1 Star public members

`int run();`

Execute the Star. This method also interfaces to the SimControl class to provide for control over simulations. All derived classes that override this method must invoke `Star::run.`

`StringList print(int` *`verbose`* `= 0) const;`

Print out info on the star.

```
Star& asStar();
const Star& asStar() const;
```

These simply return a reference to `this,` overriding `Block::asStar.`

`int index() const;`

Return the index value for this star. Index values are a feature that assists with certain schedulers; the idea is to assign a numeric index to each star at any level of a particular Universe or Galaxy.

`virtual void setTarget(Target*` *t*`);`

Set the target associated with this star.

`void noInternalState();`

Declare that this star has no internal state (This function may change to protected in future Ptolemy releases).

`int hasInternalState();`

Return `TRUE` if this star has internal state, `FALSE` if it doesn't. Useful in parallel scheduling.

3.2.2 Star protected members

`virtual void go();`

This is a method that is intended to be overridden to provide the principal action of executing this block. It is protected and is intended to be called from the `run()` member function. The separation is so that actions common to a domain can be provided in the run function, leaving the writer of a functional block to only implement `go()`.

3.3 Class Galaxy

A Galaxy is a type of Block that has an internal hierarchical structure. In particular, it contains other Blocks (some of which may also be galaxies). It is possible to access only the top-level blocks or to flatten the hierarchy and step through all the blocks, by means of the various iterator classes associated with Galaxy. While we generally define a different derived type of Star for each domain, the same kinds of Galaxy (and derived classes such as InterpGalaxy — are used in each domain. Accordingly, a Galaxy has a data member containing its associated domain (which is set to null by the constructor). PortHoles belonging to a Galaxy are, as a rule, aliased so that they refer to PortHoles of an interior Block, although this is not a requirement.

3.3.1 Galaxy public members

`void initialize();`

System initialize method. Derived Galaxies should not redefine initialize; they should write a `setup()` method to do any class-specific startup.

`virtual void preinitialize();`

Preinitialization of a Galaxy invokes preinitialization of all its member blocks. Preinitialization of the member blocks is done in two passes: the first pass preinits atomic blocks only, the second all blocks. This allows clean support of graphical recursion; for example, a HOFIfElseGr star can control a recursive reference to the current galaxy. The IfElse star is guaranteed to get control before the subgalaxy does, so it can delete the subgalaxy to stop the recursion. The second pass must preinit all blocks in case a non-atomic block adds a block to the current galaxy. `Galaxy::preinitialize` is called from `Galaxy::initialize`. (It would be somewhat cleaner to have the various schedulers invoke `preinitialize()` separately from `initialize()`, but that would require many more pieces of the system to know about preinitialization.) Because of this decision, blocks in subgalaxies will see a preinitialize call during the outer galaxy's preinitialize pass and then another one when the subgalaxy is itself initialized. Thus, blocks must act safely if preinitialized multiple times. (HOF stars generally destroy themselves when preinitialized, so they can't see extra calls.)

`void wrapup();`
System wrapup method. Recursively calls wrapup in subsystems

`void addBlock(Block&` *`b`*`,const char*` *`bname`*`);`
Add block to the galaxy and set its name.

`int removeBlock(Block&` *`b`*`);`
Remove the block *`b`* from the galaxy's list of blocks, if it is in the list. The block is not deleted. If the block was present, `1` is returned; otherwise `0` is returned.

`virtual void initState();`
Initialize states.

`int numberBlocks() const;`
Return the number of blocks in the galaxy.

`StringList print(int` *`verbose`*`) const;`
Print a description of the galaxy.

`int isItAtomic() const;`
Returns `FALSE` (galaxies are not atomic blocks).

`Galaxy& asGalaxy();`
`const Galaxy& asGalaxy() const;`
These return myself as a Galaxy, overriding `Block::asGalaxy.`

`const char* domain() const;`
Return my domain.

`void setDomain(const char*` *`dom`*`);`
Set the domain of the galaxy (this may become a protected member in the future).

`Block* blockWithName(const char*` *`name`*`);`
Support blockWithName message to access internal block list.

3.3.2 Galaxy protected members

`void addBlock(Block&` *`b`*`)`
Add *`b`* to my list of blocks.

```
void connect(GenericPort& source, GenericPort& destination,
             int numberDelays = 0)
```
Connect sub-blocks with a delay (default to zero delay).

`void alias(PortHole& galPort, PortHole& blockPort);`
`void alias(MultiPortHole& galPort, MultiPortHole& blockPort);`

Connect a Galaxy PortHole to a PortHole of a sub-block, or same for a MultiPortHole.

`void initSubblocks();`
`void initStateSubblocks();`

Former: initialize subblocks only. Latter: initialize states in subblocks only.

3.3.3 Galaxy iterators

There are three types of iterators associated with a Galaxy: GalTopBlockIter, GalAllBlockIter, and GalStarIter. The first two iterators return pointers to Block; the final one returns a pointer to Star. As its name suggests, GalTopBlockIter returns only the Blocks on the top level of the galaxy. GalAllBlockIter returns Blocks at all levels of the hierarchy, in depth-first order; if there is a galaxy inside the galaxy, first it is returned, then its contents are returned. Finally, GalStarIter returns only the atomic blocks in the Galaxy, in depth-first order. There is also a const form of GalTopBlockIter, called CGalTopBlockIter. Here is a function that prints out the names of all stars at any level of the given galaxy onto a given stream.

```
void printNames(Galaxy& g,ostream& stream) {
    GalStarIter nextStar(g);
    Star* s;
    while ((s = nextStar++) != 0)
        stream << s->fullName() << "\back n";
}
```

3.4 Class DynamicGalaxy

A DynamicGalaxy is a type of Galaxy for which all blocks, ports, and states are allocated on the heap. When destroyed, it destroys all of its blocks, ports, and states in a clean manner. There's not much more to it than that: it provides a destructor, class identification functions `isA` and `className,` and little else.

3.5 Class InterpGalaxy

InterpGalaxy is derived from DynamicGalaxy. It is the key workhorse for interfacing between user interfaces, such as ptcl or pigi, and the Ptolemy kernel, because it has commands for building structures given commands specified in the form of text strings. These commands add stars and galaxies of given types and build connections between them. InterpGalaxy interacts with the KnownBlock class to create stars and galaxies, and the Domain class to create wormholes. InterpGalaxy differs from other classes derived from Block in that the "class name" (the value returned by `className())` is a variable; the class is used to create many different "derived classes" corresponding to different topologies. In order to use InterpGalaxy to make a user-defined galaxy type, a series of commands are executed that add stars, connections, and other features to the galaxy. When a complete galaxy has been designed, the `addToKnownList` member function adds the complete object to the known list, an action that has the effect of adding a new "class" to the system. InterpGalaxy methods that return an int return `1` for success and `0` for failure. On failure, an appropriate error message is generated by means of the Error class.

3.5.1 Building structures with InterpGalaxy

The no-argument constructor creates an empty galaxy. There is a constructor that takes a single `const` char * argument specifying the class name (the value to be returned by `className().` The copy constructor creates another InterpGalaxy with the identical internal structure. There is also an assignment operator that does much the same.

`void setDescriptor(const char* `*`dtext`*`)`

Set the descriptor. Note that this is public, though the NamedObj function is protected. *`dtext`* must live as long as the InterpGalaxy does.

`int addStar(const char* `*`starname`*`, const char* `*`starclass`*`);`

Add a new star or galaxy with class name *`starclass`* to this InterpGalaxy, naming the new instance *`starname.`* The known block list for the current domain is searched to find *`starclass.`* Returns 1 on success, 0 on failure. On failures, an error message of the form

```
No star/galaxy named ’starclass’ in domain ’current-domain’
```

will be produced. The name is a misnomer since *`starclass`* may name a galaxy or a wormhole.

```
int connect(const char* srcblock, const char* srcport,
            const char* dstblock, const char* dstport,
            const char* delay = 0);
```

This method creates a point-to-point connection between the port *`srcport`* in the subblock *`srcblock`* and the port *`dstport`* in the subblock *`dstblock,`* with a delay value represented by the expression *`delay.`* If the delay parameter is omitted there is no delay. The delay expression has the same form as an initial value for an integer state (class IntState), and is parsed in the same way as an IntState belonging to a subblock of the galaxy would be. `1` is returned for success, `0` for failure. A variety of error messages relating to nonexistent blocks or ports may be produced.

```
int busConnect(const char* srcblock, const char* srcport,
               const char* dstblock, const char* dstport,
               const char* width, const char* delay = 0);
```

This method creates a point-to-point bus connection between the multiport *`srcport`* in the subblock *`srcblock`* and the multiport *`dstport`* in the subblock *`dstblock,`* with a width value represented by the expression *`width`* and delay value represented by the expression *`delay.`* If the delay parameter is omitted there is no delay. A bus connection is a series of parallel connections: each multiport contains *`width`* portholes and all are connected in parallel. The delay and width expressions have the same form as an initial value for an integer state (class IntState), and are parsed in the same way as an IntState belonging to a subblock of the galaxy would be. `1` is returned for success, `0` for failure. A variety of error messages relating to nonexistent blocks or multiports may be produced.

`int alias(const char* `*`galport`*`, const char* `*`block`*`, const char *`*`blockport`*`);`

Create a new port for the galaxy and make it an alias for the porthole *`blockport`* contained in the subblock *`block.`* Note that this is unlike the Galaxy `alias` method in that this method creates the galaxy port.

`int addNode(const char* nodename);`

Create a node for use in netlist-style connections and name it *nodename.*

```
int nodeConnect(const char* blockname, const char* portname,
                const char* node, const char* delay = 0);
```

Connect the porthole named *portname* in the subblock named *blockname* to the node named *node.* Return `1` for success, `0` and an error message for failure.

```
int addState(const char* statename, const char* stateclass,
             const char* statevalue);
```

Add a new state named *statename,* of type *stateclass,* to the galaxy. Its default initial value is given by *statevalue.*

```
int setState(const char* blockname, const char* statename,
             const char* statevalue);
```

Change the initial value of the state named *statename* that belongs to the subblock *blockname* to the string given by *statevalue.* As a special case, if *blockname* is the string `this,` the state belonging to the galaxy, rather than one belonging to a subblock, is changed.

`int setDomain(const char* newDomain);`

Change the inner domain of the galaxy to *newDomain.* This is the technique used to create wormholes (that are one domain on the outside and a different domain on the inside). It is not legal to call this function if the galaxy already contains stars.

`int numPorts(const char* blockname, const char* portname, int numP);`

Here *portname* names a multiporthole and *blockname* names the block containing it. *numP* portholes are created within the multiporthole; these become ports of the block as a whole. The names of the portholes are formed by appending `#1, #2,` etc. to the name of the multiporthole.

3.5.2 Deleting InterpGalaxy structures

`int delStar(const char* starname);`

Delete the instance named *starname* from the current galaxy. Ports of other stars that were connected to ports of *starname* will become disconnected. Returns 1 on success, 0 on failure. On failure an error message of the form

```
No instance of ‘‘starname’’ in ‘‘galaxyname’’
```

will be produced. The name is a misnomer since *starclass* may name a galaxy or a wormhole.

`int disconnect(const char* block, const char* port);`

Disconnect the porthole *port,* in subblock *block,* from whatever it is connected to. This works for point-to-point or netlist connections.

`int delNode(const char* nodename);`
`Delete the node nodename.`

3.5.3 InterpGalaxy and cloning

```
Block *makeNew() const;
Block *clone() const;
```

For InterpGalaxy the above two functions have the same implementation. An identical copy of the current object is created on the heap.

```
void addToKnownList(const char* definitionSource,
                    const char* outerDomain,
                      Target* innerTarget = 0);
```

This function adds the galaxy to the known list, completing the definition of a galaxy class. The “class name” is determined by the name of the InterpGalaxy (as set by Block::setBlock or in some other way). This class name will be returned by the className function, both for this InterpGalaxy and for any others produced from it by cloning. If *outerDomain* is different from the system’s current domain (read from class KnownBlock), a wormhole will be created. A wormhole will also be created if *innerTarget* is specified, or if galaxies for the domain *outerDomain* are always wormholes (this is determined by asking the Domain class). Once addToKnownList is called on an InterpGalaxy object, that object should not be modified further or deleted. The KnownBlock class will manage it from this point on. It will be deleted if a second definition with the same name is added to the known list, or when the program exits.

3.5.4 Other InterpGalaxy functions

```
const char* className() const
```

Return the current class name (which can be changed). Unlike most other classes, where this function returns the C++ class name, we consider the class name of galaxies built by InterpGalaxy to be variable; it is set by `addToKnownList` and copied from one galaxy to another by the copy constructor or by cloning.

```
void preinitialize();
```

Overrides `Galaxy::preinitialize()`. This re-executes initialization steps that depend on variable parameters, such as delays and bus connections for which the delay value or bus width is an expression with variables. `Galaxy::preinitialize` is then invoked to preinitialize the member blocks.

```
Block* blockWithDottedName(const char* name);
```

Returns a pointer to an inner block, at any depth, whose name matches the specification *`name`.* For example, `blockWithDottedName("a.b.c")` would look first for a subgalaxy named `"a",` then within that for a subgalaxy named `"b",` and finally with that for a subgalaxy named `"c",` returning either a pointer to the final Block or a null pointer if a match is not found.

3.6 Class Runnable

The Runnable class is a sort of “mixin class” intended to be used with multiple inheritance to create runnable universes and wormholes. It is defined in the file `Universe.h.` Constructors:

```
Runnable(Target* tar, const char* ty, Galaxy* g);
Runnable(const char* targetname, const char* dom, Galaxy* g);
```

`void initTarget();`

This function initializes target and/or generates the schedule.

`int run();`

This function causes the object to run, until the stopping condition is reached.

`virtual void setStopTime(double` *`stamp`*`);`

This function sets stop time. The default implementation just calls the identical function in the target.

`StringList displaySchedule();`

Display schedule, if appropriate (some types of schedulers will return a string saying that compile-time scheduling is not performed, e.g. DE and DDF schedulers).

`virtual ~Runnable();`

The destructor deletes the Target.

A Runnable object has the following protected data members:

```
const char* type;
Galaxy* galP;
```

As a rule, when used as one of the base classes for multiple inheritance, the `galP` pointer will point to the galaxy provided by the other half of the object.

A Runnable object has the private data member:

```
Target* target;
```

3.7 Class Universe

Class Universe is inherited from both Galaxy and Runnable. It is intended for use in standalone Ptolemy applications. For applications that use a user interface to dynamically build universes, class InterpUniverse is used instead. In addition to the Runnable and Galaxy functions, it has:

```
Universe(Target* s,const char* typeDesc);
The constructor specifies the target and the universe type.
Scheduler* scheduler() const;
```

Returns the scheduler belonging to the universe’s target.

`int run();`

Return `Runnable::Run`.

3.8 Class InterpUniverse

Class InterpUniverse is inherited from both InterpGalaxy and Runnable. Ptolemy user interfaces build and execute InterpUniverses. In addition to the standard InterpGalaxy functions, it provides:

`InterpUniverse (const char*` *`name`* `= "mainGalaxy");`

This creates an empty universe with no target and the given name. If no name is specified, `mainGalaxy` is the default.

`int newTarget(const char*` *`newTargName`* = 0);
: This creates a target of the given name (from the KnownTarget list), deleting any existing target.

`const char* targetName() const;`
: Return the name of the current target.

`Scheduler* scheduler() const;`
: Return the scheduler belonging to the current target (`0` if none).

`Target* myTarget() const;`
: Return a pointer to the current target.

`int run();`
: Invokes `Runnable::run`.

`void wrapup();`
: Invokes wrapup on the target.

Chapter 4. Control of Execution and Error Reporting

Authors: *Joseph T. Buck*

Other Contributors: *John S. Davis II*

The principal classes responsible for control of the execution of the universe are the Target and the Scheduler. The Target has high-level control over what happens when a user types *`run`* from the interface. Targets take on particular importance in code generation domains where they describe all the features of the target of execution, but they are used to control execution in simulation domains as well. Targets use Schedulers to control the order of execution of Blocks under their control. In some domains, the Scheduler does almost everything; the Target simply starts it up. In others, the Scheduler determines an execution order and the Target takes care of many other details, such as generating code in accordance with the schedule, downloading the code to an embedded processor, and executing it. The Error class provides a means to format error messages and optionally to halt execution. The interface is always the same, but different user interfaces typically provide different implementations of the methods of this class. The SimControl class provides a means to register actions for execution during a simulation, as well as facilities to cleanly halt execution on an error.

4.1 Class Target

Class Target is derived from class Block; as such, it can have states and a parent (the fact that it can also have portholes is not currently used). A Target is capable of supervising the execution of only certain types of Stars; the *`starClass`* argument in its constructor specifies what type. A Universe or InterpUniverse is run by executing its Target. Targets have Schedulers, which as a rule control order of execution, but it is the Target that is *in control*. A Target can have children that are other Targets; this is used, for example, to represent multi-processor systems for which code is being generated (the parent target represents the system as a whole, and child targets represent each processor).

4.1.1 Target public members

`Target(const char*` *`name`*, const char* *`starClass`*,const char* *`desc`* = "");

This is the signature of the Target constructor. *`name`* specifies the name of the Target and *`desc`* specifies its descriptor (these fields end up filling in the corresponding NamedObj fields). The *`starClass`* argument specifies the class of stars that can be executed by the Target. For example, specifying `DataFlowStar` for this argument means that the Target can run any type of star of this class or a class derived from it. The `isA` function is used to perform the check. See the description of `auxStarClass` below.

`const char* starType() const;`

Return the supported star class (the *`starClass`* argument from the constructor).

`Scheduler* scheduler() const;`

Return a pointer to my scheduler.

`Target* cloneTarget() const;`

This simply returns the result of the `clone` function as a Target. It is used by the KnownTarget class, for example to create a Target object corresponding to a name specified from a user interface.

`virtual StringList displaySchedule();`

The default implementation simply passes this call along to the scheduler; derived classes may modify this.

`virtual StringList pragma () const;`

A Target may understand certain annotations associated with Blocks called *pragmas*. For example, an annotation may specify how many times a particular Star should fire. Or it could specify that a particular Block should be mapped onto a particular processor. Or it could specify that a particular State of a particular Block should be settable on the command line that invokes a generated program. The above method returns the list of named pragmas that a particular target understands (e.g. *`firingsPerIteration`* or *`processorNumber`*). In derived classes, each item in the list is a three part string, "*`name type value`*", separated by spaces. The *`value`* will be a default value. The implementation in class Target returns a StringList with only a single zero-length string in it. The *`type`* can be any type used in states.

`virtual StringList pragma (const char*` *`parentname`*,
    `const char*` *`blockname`*) `const;`

To determine the value of all pragmas that have been specified for a particular block, call this method. In derived classes, it returns a list of "*`name value`*" pairs, separated by spaces. In the base class, it returns an empty string. The *`parentname`* is the name of the parent class (universe or galaxy master name).

`virtual StringList pragma (const char*` *`parentname`*,
`const char*` *`blockname`*,
`const char*` *`pragmaname`*) `const;`

To determine the value of a pragma of a particular type that has been specified for a particular block, call this method. In derived classes, it returns a value. In the base class, it returns a zero-length string.

`virtual StringList pragma (const char*` *`parentname`*,
`const char*` *`blockname`*,
`const char*` *`pragmaname`*,
`const char*` *`value`*) `const;`

To specify a pragma to a target, call this method. The implementation in the base class “Target” does nothing. In derived classes, the pragma will be registered in some way. The return value is always a zero-length string.

`Target* child(int` *`n`*`);`
: Return the *`nth`* child Target, null if no children or if *`n`* exceeds the number of children.

`Target* proc(int` *`n`*`);`
: This is the same as `child` if there are children. If there are no children, an argument of 0 will return a pointer to the object on which it is called, otherwise a null pointer is returned.

`int nProcs() const;`
: Return the number of processors (1 if no children, otherwise the number of children).

`virtual int hasResourcesFor(Star&` *`s`*`,const char*` *`extra`*`=0);`
: Determine whether this target has the necessary resources to run the given star. It is virtual in case this is necessary in child classes. The default implementation uses *`resources`* states of the target and the star.

`virtual int childHasResources(Star&` *`s`*`,int` *`childNum`*`);`
: Determine whether a particular child target has resources to run the given star. It is virtual in case later necessary.

`virtual void setGalaxy(Galaxy&` *`g`*`);`
: Associate a Galaxy with the Target. The default implementation just sets its galaxy pointer to point to *`g.`*

`virtual void setStopTime(double` *`when`*`);`
: Set the stopping condition. The default implementation just passes this on to the scheduler.

`virtual void resetStopTime(double` *`when`*`);`
: Reset the stopping condition for the wormhole containing this Target. The default implementation just passes this on to the scheduler. In addition to the action performed by `setStopTime,` this function also does any synchronization required by wormholes.

`virtual void setCurrentTime(double` *`now`*`);`
: Set the current time to *`now.`*

```
virtual int run();
virtual void wrapup();
```

The following methods are provided for code generation; schedulers may call these. They may move to class CGTarget in a future Ptolemy release.

`virtual void beginIteration(int` *`repetitions`*`, int` *`depth`*`);`
: Function called to begin an iteration (default version does nothing).

`virtual void endIteration(int` *`repetitions`*`, int` *`depth`*`);`
: Function called to end an iteration (default version does nothing).

`virtual void writeFiring(Star&` *`s`*`, int` *`depth`*`);`
: Function called to generate code for the star, with any modifications required by this par-

ticular Target (the default version does nothing).

`virtual void beginIf(PortHole&` *`cond`*, int *`truthdir`*,
`int` *`depth`*, int *`haveElsePart`*);
`virtual void beginElse(int` *`depth`*);
`virtual void endIf(int` *`depth`*);
`virtual void beginDoWhile(PortHole&` *`cond`*, int *`truthdir`*, int *`depth`*);
`virtual void endDoWhile(PortHole&` *`cond`*);

These above functions are used in code generation to generate conditionals. The default implementations do nothing.

`virtual int commTime(int` *`sender`*,int *`receiver`*,int *`nUnits`*, int *`type`*);

Return the number of time units required to send *`nUnits`* units of data whose type is the code indicated by *`type`* from the child Target numbered *`sender`* to the child target numbered *`receiver.`* The default implementation returns 0 regardless of the parameters. No meaning is specified at this level for the type codes, as different languages have different types; all that is required is that different types supported by a particular target map into distinct type codes.

`Galaxy* galaxy();`

Return my galaxy pointer (0 if it has not been set).

4.1.2 Target protected members

`virtual void setup();`

This is the main initialization function for the target. It is called by the `initialize` function, which by default initializes the Target states. The default implementation calls `galaxySetup(),` and if it returns a nonzero value, then calls `schedulerSetup().`

`virtual int galaxySetup();`

This method (and overloaded versions of it) is responsible for checking the galaxy belonging to the target. In the default implementation, each star is checked to see if its type is supported by the target (because the `isA` function reports that it is one of the supported star classes). If a star does not match this condition an error is reported. In addition, `setTarget()` is called for each star with a pointer to the Target as an argument. If there are errors, 0 is returned, otherwise 1.

`virtual int schedulerSetup();`

This method (and overloaded versions of it) are responsible for initializing an execution of the universe. The default implementation initializes the scheduler and calls `setup()` on it.

`void setSched(Scheduler*` *`sch`*);

The target's scheduler is set to *`sch,`* which must either point to a scheduler on the heap or be a null pointer. Any preexisting scheduler is deleted. Also, the scheduler's `setTarget` member is called, associating the Target with the Scheduler.

`void delSched();`

This function deletes the target's scheduler and sets the scheduler pointer to null.

`void addChild(Target&` *`child`*`);`

Add *`child`* as a child target.

`void inheritChildren(Target*` *`parent`*`, int` *`start`*`, int` *`stop`*`);`

This method permits two different Target objects to share child Targets. The child targets numbered *`start`* through *`stop`* of the Target pointed to by *`parent`* become the children of this Target (the one on which this method is called). Its primary use is in multi-processor scheduling or code generation, in which some construct is assigned to a group of processors. It has a big disadvantage; the range of child targets must be continuous.

`void remChildren();`

Remove the *`children`* list. This does not delete the child targets.

`void deleteChildren();`

Delete all the *`children`*. This assumes that the child Targets were created with `new.`

`virtual const char* auxStarClass() const;`

Auxiliary star class: permits a second type of star in addition to the supported star class (see `startType()`). The default implementation returns a null pointer, indicating no auxiliary star class. Sorry, there is no present way to support yet a third type.

`const char* writeDirectoryName(const char*` *`dirName`* `= 0);`

This method returns a directory name that is intended for use in writing files, particularly for code generation targets. If the directory does not exist, it attempts to create it. Returns the fully expanded path name (which is saved by the target).

`const char* workingDirectory() const;`

Return the directory name previously set by `writeDirectoryName.`

`char* writeFileName(const char*` *`fileName`* `= 0);`

Method to set a file name for writing. *`writeFileName`* prepends *`dirFullName`* (which was set by `writeDirectoryName)` to *`fileName`* with "`/`" between. Always returns a pointer to a string in new memory. It is up to the user to delete the memory when no longer needed. If *`dirFullName`* or *`fileName`* is NULL then it returns a pointer to a new copy of the string `/dev/null`.

4.2 Class Scheduler

Scheduler objects determine the order of execution of Stars. As a rule, they are created and managed by Targets. Some schedulers, such as those for the SDF domain, completely determine the order of execution of blocks before any blocks are executed; others, such as those for the DE domain, supervise the execution of blocks at run time. The Scheduler class is an abstract base class; you can’t have an object of class Scheduler. All schedulers have a pointer to the Target that controls them as well as to a Galaxy. Usually the Galaxy will be the same one that the Target points to, but this is not a requirement. The Scheduler constructor just zeros its target, galaxy pointers. The destructor is virtual and do-nothing.

4.2.1 Scheduler public members

`virtual void setGalaxy(Galaxy& g);`

This function sets the galaxy pointer to point to `g`.

`Galaxy* galaxy();`

This function returns the galaxy pointer.

`virtual void setup() = 0;`

This function (in derived classes) sets up the schedule. In compile-time schedulers such as those for SDF, a complete schedule is computed; others may do little more than minimal checks.

`virtual void setStopTime(double limit) = 0;`

Set the stop time for the scheduler. Schedulers have an abstract notion of time; this determines how long the scheduler will run for.

`virtual double getStopTime() = 0;`

Retrieve the stop time.

`virtual void resetStopTime(double limit);`

Reset the stopping condition for the wormhole containing this Scheduler. The default implementation simply calls `setStopTime` with the same argument. For some derived types of schedulers, additional actions will be performed as well by derived Scheduler classes.

`virtual int run() = 0;`

Run the scheduler until the stop time is reached, an error condition occurs, or it stops for some other reason.

`virtual void setCurrentTime(double val);`

Set the current time for the scheduler.

`virtual StringList displaySchedule();`

Return the schedule if this makes sense.

`double now() const;`

Return the current time (the value of the protected member `currentTime`).

`int stopBeforeDeadlocked() const;`

Return the value of the `stopBeforeDeadFlag` protected member. It is set in timed domains to indicate that a scheduler inside a wormhole was suspended even though it had more work to do.

`virtual const char* domain() const;`

Return the domain for this scheduler. This method is no longer used and will be removed from future releases; it dates back to the days in which a given scheduler could only be used in one domain.

`void setTarget(Target& `*`t`*`);`
: Set the target pointer to point to *`t`*.

`Target& target ();`
: Return the target.

`virtual void compileRun();`
: Call code-generation functions in the Target to generate code for a run. In the base class, this just causes an error.

The following functions now forward requests to SimControl, which is responsible for controlling the simulation.

`static void requestHalt();`
: Calls `SimControl::declareErrorHalt.` NOTE: `SimControl::requestHalt` only sets the halt bit, not the error bit.

`static int haltRequested();`
: Calls `SimControl::haltRequested.` Returns TRUE if the execution should halt.

`static void clearHalt();`
: Calls `SimControl::clearHalt.` Clears the halt and error bits.

4.2.2 Scheduler protected members

The following two data members are protected.

```
// current time of the scheduler
double currentTime;
// flag set if stop before deadlocked.
// for untimed domain, it is always FALSE.
int stopBeforeDeadlocked;
```

4.3 Class Error

Class Error is used for error reporting. While the interfaces to these functions are always the same, different user interfaces provide different implementations: `ptcl` connects to the Tcl error reporting mechanism, `pigi` pops up windows containing error messages, and `interpreter` simply prints messages on the standard error stream. All member functions of Error are static. There are four "levels" of messages that may be produced by the error facility: `Error::abortRun` is used to report an error and cause execution of the current universe to halt. `Error::error` reports an error. `Error::warn` reports a warning, and `Error::message` prints an information message that is not considered an error. Each of these four functions is available with two different signatures. For example:

```
static void abortRun (const char*, const char* = 0, const char* = 0);
static void abortRun (const NamedObj& obj, const char*, const char* = 0,
                      const char* = 0);
```

The first form produces the error message by simply concatenating its arguments (the second and third arguments may be omitted); no space is added. The second form prepends the full name of the *`obj`* argument, a colon, and a space to the text provided by the remaining arguments. If the implementation provides a marking facility, the object named by *`obj`* is marked by the user interface (at present, the interface associated with `pigi` will highlight the object if its icon appears on the screen). The remaining static Error functions `error`, `warn`, and `message` have the same signatures as does `abortRun` (there are the same two forms for each function). In addition, the Error class provides access to the marking facility, if it exists:

`static int canMark();`

This function returns TRUE if the interface can mark NamedObj objects (generally true for graphic interfaces), and FALSE if it cannot (generally true for text interfaces).

`static void mark (const NamedObj&` *`obj`*`);`

This function marks the object *`obj`*, if marking is implemented for this interface. It is a no-op if marking is not implemented.

4.4 Class SimControl

The SimControl class controls execution of the simulation. It has some global status flags that indicate whether there has been an error in the execution or if a halt has been requested. It also has mechanisms for registering functions to be called before or after star executions, or in response to a particular star's execution, and responding to interrupts. This class interacts with the Error class (which sets error and halt bits) and the Star class (to permit execution of registered actions when stars are fired). Schedulers and Targets are expected to monitor the SimControl halt flag to halt execution when errors are signaled and halts are requested. Once exceptions are commonplace in C++ implementations, a cleaner implementation could be produced.

4.4.1 Access to SimControl status flags.

SimControl currently has four global status bits: the error bit, the halt bit, the interrupt bit, and the poll bit. These functions set, clear, or report on these bits.

`static void requestHalt ();`

This function sets the halt bit. The effect is to cause schedulers and targets to cease execution. It is important to note that this function does not alter flow of control; it only sets a flag.

`static void declareErrorHalt ();`

This is the same as `requestHalt` except that it also sets the error bit. It is called, for example, by `Error::abortRun.`

`static int haltRequested ();`

This function returns true if the halt bit is set, false otherwise. If the poll or interrupt bits are set, it calls handlers for them (see the subsection describing these).

`static void clearHalt ();`

This function clears the halt and error flags.

4.4.2 Pre-actions and Post-actions

SimControl can register a function that will be called before or after the execution of a particular star, or before or after the execution of all stars. A function that is called before a star is a *preaction*; on that is called after a star is a *post-action*. The functions that can be registered have take two arguments: a pointer to a Star (possibly null), and a `const` char* pointer that points to a string (possibly null). The type definition

```
typedef void (*SimActionFunction)(Star*,const char*);
```

gives the name SimActionFunction to functions of this type; several SimControl functions take arguments of this form.

```
static SimAction* registerAction(SimActionFunction action, int pre,
 const char* textArg = 0, Star* which = 0);
```

Register a pre-action or post-action. If *`pre`* is TRUE it is a preaction. If *`textArg`* is given, it is passed as an argument when the action function is called. If *`which`* is 0, the function will be called unconditionally by `doPreActions` (if it is a preaction) or `doPostActions` (if it is a post-action; otherwise it will only be called if the star being executed has the same address as `which.` The return value represents the registered action; class SimAction is treated as if it is opaque (I'm not telling you what is in it) which can be used for `cancel` calls.

```
static int doPreActions(Star * which);
static int doPostActions(Star * which);
```

Execute all pre-actions, or post-actions, for a the particular Star *`which.`* The *`which`* pointer is passed to each action function, along with any text argument declared when the action was registered. Return TRUE if no halting condition arises, FALSE if we are to halt.

```
static int cancel(SimAction* action);
```

Cancel *`action.`* Warning: argument is deleted. Future versions will provide more ways of cancelling actions.

4.4.3 SimControl interrupts and polling

Features in this section will be used in a new graphic interface; they are mostly untested at this point. The SimControl class can handle interrupts and can register a polling function that is called for every star execution. It only provides one handler.

```
static void catchInt(int signo = -1, int always = 0);
```

This static member function installs a simple interrupt handler for the signal with Unix signal number *`signo.`* If *`always`* is true, the signal is always caught; otherwise the signal is not caught if the current status of the signal is that it is ignored (for example, processes running in the background ignore interrupt signals from the keyboard). This handler simply sets the SimControl interrupt bit; on the next call to `haltRequested,` the user-specified interrupt handler is called.

```
static SimHandlerFunction setInterrupt(SimHandlerFunction f);
```

Set the user-specified interrupt handler to *`f,`* and return the old handler, if any. This function is called in response to any signals specified in `catchInt.`

`static SimHandlerFunction setPoll(SimHandlerFunction f);`

Register a function to be called by `haltRequested` if the poll flag is set, and set the poll flag. Returns old handler if any.

Chapter 5. Interfacing domains – wormholes and related classes

Authors: *Joseph T. Buck*

Other Contributors: *J. Liu*

This section describes the classes that implement the mechanism that allows different domains to be interfaced. It is this ability to integrate different domains that sets Ptolemy apart from other systems.

5.1 Class Wormhole

A wormhole for a domain is much like a star belonging to that domain, but it contains pointers to a subsystem that operates in a different domain. The interface to that other domain is through a “universal event horizon”. The wormhole design, therefore, does not depend on the domain it contains, but only on the domain in which it is used as a block. It must look like a star in that outer domain. The base Wormhole class is derived from class Runnable , just like the class Universe . Every member of the Runnable class has a pointer to a component Galaxy and a Target (\pxref class Target). Like a Universe, a Wormhole can perform the scheduling actions on the component Galaxy. A Wormhole is different from a Universe in that it is not a stand-alone object. Instead, it is triggered from the outer domain to initiate the scheduling. Also, since Wormhole is an abstract base class, you cannot create an object of class Wormhole; only derived Wormholes can be created. Each domain has a derived Wormhole class. For example, the SDF domain has class SDFWormhole. This domain-specific Wormhole is derived from not only the base Wormhole class but also from the domain-specific star class, SDFStar. This multiple inheritance realizes the inherent nature of the Wormhole. First, the Wormhole behaves exactly like a Star from the outer domain (SDF) since it is derived from SDFStar. Second, internally it can encapsulate an entire foreign domain with a separate Galaxy and a separate Target and Scheduler.

5.1.1 Wormhole public members

`const char* insideDomain() const;`

This function returns the name of the inside domain.

`void setStopTime(double` *`stamp`*`);`

This function sets the stop time for the inner universe.

`Wormhole(Star&` *`self`*`, Galaxy&` *`g`*`, const char*` *`targetName`* `= 0);`
`Wormhole(Star&` *`self`*`, Galaxy&` *`g`*`, Target*` *`innerTarget`* `= 0);`

The above two signatures represent the constructors provided for class Wormhole. We never use plain Wormholes; instead we always have objects derived from Wormhole and some kind of Star. For example:

```
class SDFWormhole : public Wormhole, public SDFStar {
public:
    SDFWormhole(Galaxy& g,Target* t) : Wormhole(*this,g,t) {
        buildEventHorizons();
    }
};
```

The first argument to the constructor should always be a reference to the object itself, and represents “the wormhole as a star”. The second argument is the inner galaxy. The third argument describes the target of the Wormhole, and may be provided either as a Target object or by name, in which case it is created by using the KnownTarget class.

`Scheduler* outerSched();`

This returns a pointer to the scheduler for the outer domain (the one that lives above the wormhole). The scheduler for the inner domain for derived wormhole classes can be obtained from the `scheduler()` method.

5.1.2 Wormhole protected members

`void setup();`

The default implementation calls `initTarget.`

`int run();`

This function executes the inside of the wormhole for the appropriate amount of time.

`void buildEventHorizons ();`

This function creates the EventHorizon objects that connect the inner galaxy ports to the outside. A pair of EventHorizons is created for each galaxy port. It is typically called by the constructor for the XXXWormhole, where XXX is the outer domain name.

`void freeContents();`

This function deletes the event horizons and the inside galaxy. It is intended to be called from XXXWormhole destructors. It cannot be part of the Wormhole constructor due to an ordering problem (we want to assure that it is called before the destructor for either of XXXWormhole’s two base classes is called).

`virtual double getStopTime() = 0;`

Get the stopping condition for the inner domain. This is a pure virtual function and must be redefined in the derived class.

`virtual void sumUp();`

This function is called by `Wormhole::run` after running the inner domain. The default implementation does nothing. Derived wormholes can redefine it to put in any “summing up” work that is required after running the inner domain.

`Galaxy& gal;`

The member `gal` is a reference to the inner galaxy of the Wormhole.

5.2 Class EventHorizon

Class EventHorizon is another example of a “mixin class”; EventHorizon has the same relationship to PortHoles as Wormhole has to Stars. The name is chosen from cosmology, representing the point at which an object disappears from the outside universe and enters the interior of a black hole, which can be thought of as a different universe entirely. As for wormholes, we never consider objects that are “just an EventHorizon”. Instead, all objects that are actually used are multiply inherited from EventHorizon and from some type of PortHole class. For each type of domain we require two types of EventHorizon. The first, derived from ToEventHorizon, converts from a format suitable for a particular domain to the “universal form”. The other, derived from FromEventHorizon, converts from the universal form to the domain-specific form.

5.2.1 How EventHorizons are used

Generally, EventHorizons are used in pairs to form a connection across a domain boundary between domain XXX and domain YYY. An object of class XXXToUniversal (derived from XXXPortHole and ToEventHorizon) and an object of class YYYFromUniversal (derived from YYYPortHole and FromEventHorizon) are inserted between the ordinary, domain-specific PortHoles. The `far()` member of the XXXToUniversal points to the XXXPortHole; the `ghostAsPort()` member points to the YYYFromUniversal object. Similarly, for the YYY-FromUniveral object, `far()` points to the YYYPortHole and `ghostAsPort()` points to the XXXToUniversal object. These pairs of EventHorizons are created by the `buildEventHorizons` member function of class Wormhole.

5.2.2 EventHorizon public members

`EventHorizon(PortHole*` *`self`*`);`

The constructor for EventHorizon takes one argument, representing (for derived classes that call this constructor from their own), “myself” as a PortHole (a pointer to the PortHole part of the object). The destructor is declared virtual and does nothing.

`PortHole* asPort();`

This returns “myself as a PortHole”.

`PortHole* ghostAsPort();`

This returns a pointer to the “matching event horizon” as a porthole.

`virtual void ghostConnect(EventHorizon&` *`to`*`);`

This connects another EventHorizon to myself and makes it my “ghost port”.

`virtual int isItInput() const;`
`virtual int isItOutput() const;`

Say if I am an input or an output.

`virtual int onlyOne() const;`

Derived EventHorizon classes should redefine this method to return `TRUE` for domains in which only one particle may cross the event horizon boundary per execution. The default implementation returns `FALSE`.

`virtual void setEventHorizon(inOutType` *`inOut`*`, const char*` *`portName`*`,`
`Wormhole*` *`parentWormhole`*`, Star*` *`parentStar`*`,`
`DataType` *`type`* `= FLOAT, unsigned` *`numTokens`* `= 1 );`

Sets parameters for the EventHorizon.

`double getTimeMark();`
`void setTimeMark(double` *`d`*`);`

Get and set the time mark. The time mark is an internal detail used for bookkeeping by schedulers.

```
virtual void initialize();
Scheduler *innerSched();
Scheduler *outerSched();
```

These methods return a pointer to the scheduler that lives inside the wormhole, or outside the wormhole, respectively.

5.2.3 EventHorizon protected members

`void moveFromGhost(EventHorizon&` *`from`*`, int` *`numParticles`*`);`

Move *`numParticles`* from the buffer of *`from,`* another EventHorizon, to mine (the object on which this function is called). This is used to implement `ToEventHorizon::transferData.`

`CircularBuffer* buffer();`

Access the myBuffer of the porthole.

`EventHorizon* ghostPort;`

This is the peer event horizon.

`Wormhole* wormhole;`

This points to the Wormhole I am a member of.

```
int tokenNew;
double timeMark;
```

TimeMark of the current data, which is necessary for interface of two domains. This may become a private member in future versions of Ptolemy.

5.3 Class ToEventHorizon

A ToEventHorizon is responsible for converting from a domain-specific representation to a universal representation. It is derived from EventHorizon.

`ToEventHorizon(PortHole*` *`p`*`);`

The constructor simply calls the base class constructor, passing along its argument.

`void initialize();`

The initialize function prepares the object for execution.

`void getData();`

This protected member transfers data from the outside to the universal event horizon (myself).

`void transferData();`
This protected member transfers data from myself to my peer FromEventHorizon (the ghostPort).

5.4 Class FromEventHorizon

A FromEventHorizon is responsible for converting from a universal representation to a domain-specific representation. It is derived from EventHorizon.

`FromEventHorizon(PortHole*` *p*);
The constructor simply calls the EventHorizon constructor.

`void initialize();`
The initialize function prepares the object for execution.

`void putData();`
This protected member transfers data from Universal EventHorizon to outside.

`void transferData();`
This protected member transfers data from peer event horizon to me.

`virtual int ready();`
This is a protected member. By default, it always returns `TRUE` (`1`). Derived classes have it return `TRUE` if the event horizon is ready (there is enough data for execution to proceed), and `FALSE` otherwise.

5.5 Class WormMultiPort

The class WormMultiPort, which is derived from MultiPortHole , exists to handle the case where a galaxy with a multiporthole is embedded in a wormhole. Its `newPort` function correctly creates a pair of EventHorizon objects when a new port is created in the multiporthole. Instances of this object are created by `Wormhole::buildEventHorizons` when the inner galaxy has one or more MultiPortHole objects. Its `newConnection` method always calls `newPort.`

Chapter 6. Classes for connections between blocks

Authors: *Joseph T. Buck*

Other Contributors: *Tom Lane*
Yuhong Xiong

This chapter describes the classes that implement connections between blocks. For simulation domains, these classes are responsible for moving objects called Particles from one Block to another. For code generation domains, the Particles typically only move during scheduling and these objects merely provide information on the topology. Currently, class PortHole is also responsible for the type resolution algorithm that assigns specific types to `ANYTYPE` portholes. It would probably be better to put that function in Geodesic, which would make it simpler to provide domain-specific type resolution rules. This improvement must await a redesign of the PortHole/Geodesic structure.

6.1 Class GenericPort

The class GenericPort is a base class that provides common elements between class PortHole and class MultiPortHole. Any GenericPort object can be assumed to be either one or the other; we recommend avoiding deriving any new objects directly from GenericPort. GenericPort is derived from class NamedObj . GenericPort provides several basic facilities: aliases, which specify that another GenericPort should be used in place of this port, types, which specify the type of data to be moved by the port, and typePort, which specifies that this port has the same type as another port. When a GenericPort is destroyed, any alias or typePort pointers are automatically cleaned up, so that other GenericPorts are never left with dangling pointers. The `type()` and `typePort()` functions belong to GenericPort, not PortHole, because multiportholes have a declared type and can be type-equivalenced to other portholes. However, type resolution is strictly a PortHole notion. Multiportholes need no resolved type because they do not themselves transport particles, and indeed the concept would be ambiguous since the member ports of a multiporthole might have different resolved types. The declared type of a multiporthole is automatically assigned to its children, and its children are automatically brought into any type equivalence set the multiporthole is made part of. Thereafter, type resolution considers only the member portholes and not the multiporthole itself.

6.1.1 GenericPort query functions

```
virtual int isItInput () const;
virtual int isItOutput () const;
virtual int isItMulti () const;
```

Each of the above functions returns `TRUE` (1) or `FALSE` (0).

`StringList print (int` *`verbose`* `= 0) const;`

Print human-readable information on the GenericPort.

`DataType type () const;`

Return my DataType. This may be one of the DataType values associated with Particle classes, or the special type '`ANYTYPE`', which indicates that the type must be resolved during setup. Note that `type()` returns the port's declared type, as supplied to `setPort()`. This is not necessarily the datatype that will be chosen to pass through the port at runtime. That type is available from the `PortHole::resolvedType()` function.

`GenericPort* alias() const;`

Return my alias, or a null pointer if I have no alias. Generally, Galaxy portholes have aliases and Star portholes do not, but this is not a strict requirement.

`GenericPort* aliasFrom() const;`

Return the porthole that I am the alias for (a null pointer if none). It is guaranteed that if gp is a pointer to GenericPort and if `gp->alias()` is non-null, then the boolean expression
`gp->alias()->aliasFrom() == gp`
is always true.

`bitWord attributes() const;`

Return my attributes. Attributes are a series of bits.

`GenericPort& realPort();`
`const GenericPort& realPort() const;`

Return the real port after resolving any aliases. If I have no alias, then a reference to myself is returned.

`GenericPort* typePort() const;`

Return another generic port that is constrained to have the same type as me (0 if none). If a non-null value is called, successive calls will form a circular linked list that always returns to its starting point; that is, the loop

```
void printLoop(GenericPort& g) {
         if (g->typePort()) {
                GenericPort* gp = g;
                while (gp->typePort() != g) {
                        cout << gp->fullName() << "\back n";
                        gp = gp->typePort();
                }
        }
}
```

is guaranteed to terminate and not to dereference a null pointer.

`inline int hidden(const GenericPort&` *p*)

IMPORTANT: `hidden` is not a member function of GenericPort, but is a "plain function". It returns `TRUE` if the port in question has the `HIDDEN` attribute.

6.1.2 Other GenericPort public members

`virtual PortHole& newConnection();`

Return a reference to a porthole to be used for new connections. Class PortHole uses this one unchanged; MultiPortHole has to create a new member PortHole.

`GenericPort& setPort(const char*` *`portName`*, `Block*` *`blk`*, `DataType` *`typ`*`=FLOAT);`

Set the basic PortHole parameters: the name, parent, and data type.

`void inheritTypeFrom(GenericPort&` *`p`*`);`

Link to another port for determining the type of ‘`ANYTYPE`’ connections. The "inheritance" relationship is actually a completely symmetric constraint, and so this function would have been better named `sameTypeAs()`. Any number of portholes can be tied together by `inheritTypeFrom()` calls. Internally this is represented by chaining all the members of such a type equivalence set into a circular loop, which can be walked via `typePort()` calls. If a multiporthole is made part of a type equivalence set, all its current and future children become part of the set automatically.

```
virtual void connect(GenericPort& destination,int numberDelays,
                   const char* initDelayValues = 0);
```

Connect me with the indicated peer.

`bitWord setAttributes(const Attribute&` *`attr`*`);`

Set my attributes (some bits are turned on and others are turned off).

`void setAlias (GenericPort&` *`gp`*`);`

Set gp to be my alias. The aliasFrom pointer of gp is set to point to me.

6.1.3 GenericPort protected members

`GenericPort* translateAliases();`

The above is a protected function. If this function is called on a port with no alias, the address of the port itself is returned; otherwise, `alias()->translateAliases()` is returned.

6.2 Class PortHole

PortHole is the means that Blocks use to talk to each other. It is derived from GenericPort; as such, it has a type, an optional alias, and is optionally a member of a ring of ports of the same type connected by `typePort` pointers. It guarantees that `alias()` always returns a PortHole. In addition, a PortHole has a peer (another port that it is connected to, which is returned by `far()`), a Geodesic (a path along which particles travel between the PortHole and its peer), and a Plasma (a pool of particles, all of the same type). In simulation domains, during the execution of the simulation objects known as Particles traverse a circular path: from an output porthole through a Geodesic to an input porthole, and finally to a Plasma, where they are recirculated back to the input porthole. Like all NamedObj-derived objects, a PortHole has a parent Block. It may also be a member of a MultiPortHole, which is a logical group of PortHoles.

6.2.1 PortHole public members

The constructor sets just about everything to null pointers. The destructor disconnects the PortHole, and if there is a parent Block, removes itself from the parent's porthole list.

`PortHole& setPort(const char*` *`portName`*`, Block*` *`parent`*`,`
`DataType` *`type`* `= FLOAT);`

This function sets the name of the porthole, its parent, and its type.

`void initialize();`

This function is responsible for initializing the internal buffers of the porthole in preparation for a run.

`virtual void disconnect(int` *`delGeo`* `= 1);`

Remove a connection, and optionally attempt to delete the geodesic. The is set to zero when the geodesic must be preserved for some reason (for example, from the Geodesic's destructor). The Geodesic is deleted only if it is "temporary"; we do not delete "persistent" geodesics when we disconnect them.

`PortHole* far() const;`

Return the PortHole we are connected to.

`void setAlias (PortHole&` *`blockPort`*`);`

Set my alias to *`blockPort.`*

`int atBoundary() const;`

Return `TRUE` if this PortHole is at the wormhole boundary (if its peer is an inter-domain connection); `FALSE` otherwise.

`virtual EventHorizon* asEH();`

Return myself as an EventHorizon, if I am one. The base class returns a null pointer. EventHorizon objects (objects multiply inherited from EventHorizon and some type of PortHole) will redefine this appropriately.

`virtual void receiveData();`

Used to receive data in derived classes. The default implementation does nothing.

`virtual void sendData();`

Used to send data in derived classes. The default implementation does nothing.

`Particle& operator % (int` *`delay`*`);`

This operator returns a reference to a Particle in the PortHole's buffer. A *`delay`* value of 0 returns the "newest" particle. In dataflow domains, the argument represents the delay associated with that particular particle.

`DataType resolvedType () const;`

Return the data type computed by 'PortHole::initialize' to resolve type conversions. For example, if an `INT` output porthole is connected to a `FLOAT` input porthole, the resolved type (the type of the Particles that travel between the ports) will be `FLOAT`. Two connected portholes will always return the same resolvedType. A null pointer will be returned if the type has not yet been resolved, e.g. before initialization.

`DataType preferredType () const;`

Return the "preferred" type of the porthole. This is the same as the declared type (`GenericPort::type()`) if the declared type is not `ANYTYPE`. If the declared type is `ANYTYPE`, the preferredType is the type of the connected porthole or type equivalence set from which the `ANYTYPE`'s true type was determined. (If preferredType and resolvedType are not the same, the need for a run-time type conversion is indicated. Code generation domains may choose to splice in type conversion stars to ensure that preferredType and resolvedType are the same at all ports.) A null pointer will be returned if the type has not yet been resolved, e.g. before initialization.

`int numXfer() const;`

Returns the nominal number of tokens transferred per execution of the PortHole. It returns the value of the protected member `numberTokens.`

`int numTokens() const;`

Returns the number of particles on my Geodesic.

`int numInitDelays() const;`

Returns the number of initial delays on my Geodesic (the initial tokens, strictly speaking, are only delays in dataflow domains).

`Geodesic* geo();`

Return a pointer to my Geodesic.

`int index() const;`

Return the index value. This is a mechanism for assigning all the portholes in a universe a unique integer index, for use in table-driven schedulers.

`MultiPortHole* getMyMultiPortHole() const;`

Return the MultiPortHole that spawned this PortHole, or `NULL` if there is no such MultiPortHole.

`virtual void setDelay (int` *`newDelayValue`*`);`

Set the delay value for the connection.

`virtual Geodesic* allocateGeodesic();`

Allocate a return a Geodesic compatible with this type of PortHole. This may become a protected member in future Ptolemy releases.

`void enableLocking(const PtGate& master);`

Enable locking on access to the Geodesic and Plasma. This is appropriate for connections that cross thread boundaries. Assumption: `initialize()` has been called.

`void disableLocking();`

The converse.

`int isLockEnabled() const;`
Returns the lock status.

6.2.2 PortHole protected members

`Geodesic* myGeodesic;`
My geodesic, which connects to my peer. Initialized to `NULL`.

`PortHole* farSidePort;`
The port on the far side of the connection. `NULL` for disconnected ports.

`Plasma* myPlasma;`
Pointer to the Plasma where we get our Particles or replace unused Particles. Initialized to `NULL`.

`CircularBuffer* myBuffer;`
Buffer where the Particles are stored. This is actually a buffer of pointers to Particles, not to Particles themselves.

`int bufferSize;`
This gives the size of the CircularBuffer to allocate.

`int numberTokens;`
Number of Particles stored in the buffer each time the Geodesic is accessed. Normally this is one except for dataflow-type stars, where it is the number of Particles consumed or generated.

`void getParticle();`
Get `numberTokens` particles from the Geodesic and move them into my CircularBuffer. Actually, only Particles move. The same number of existing Particles are returned to their Plasma, so that the total number of Particles contained in the buffer remains constant.

`void putParticle();`
Move `numberTokens` particles from my CircularBuffer to the Geodesic. Replace them with the same number of Particles from the Plasma.

`void clearParticle();`
Clear `numberTokens` particles in the CircularBuffer. Leave the buffer position pointing to the last one.

`virtual int allocatePlasma();`
Allocate Plasma (default method uses global Plasma).

`int allocateLocalPlasma();`
Alternate function allocates a local Plasma (for use in derived classes).

`void deletePlasma();`
Delete Plasma if local; detach other end of connection from Plasma as well.

`void allocateBuffer();`

Allocate new buffer.

`DataType SetPreferredType();`

Function to determine preferred types during initialization. Returns the preferred type of this porthole, or 0 on failure. Protected, not private, so that subclasses that override `set-ResolvedType()` can call it.

6.2.3 CircularBuffer – a class used to implement PortHole

This class is misnamed; it is not a general circular buffer but rather an array of pointers to Particle that is accessed in a circular manner. It has a pointer representing the current position. This pointer can be advanced or backed up; it wraps around the end when this is done. The class also has a facility for keeping track of error conditions. The constructor takes an integer argument, the size of the buffer. It creates an array of pointers of that size and sets them all to null. The destructor returns any Particles in the buffer to their Plasma and then deletes the buffer.

`void reset();`

Set the access pointer to the beginning of the buffer.

`void initialize();`

Zero out the contents of the buffer.

`Particle** here() const;`

Return the access pointer. Note the double indirection; since the buffer contains pointers to Particles, the buffer pointer points to a pointer.

`Particle** next();`

Advance the pointer one position (circularly) and return the new value.

`Particle** last();`

Back up the pointer one position (circularly) and return the new value.

`void advance(int` *`n`*`);`

Advance the buffer pointer by *n* positions. This will not work correctly if *n* is larger than the buffer size. *n* is assumed positive.

`void backup(int` *`n`*`);`

Back up the buffer pointer by *n* positions. This will not work correctly if *n* is larger than the buffer size. *n* is assumed positive.

`Particle** previous(int` *`offset`*`) const;`

Find the position in the buffer *`offset`* positions in the past relative to the current position. The current position is unchanged. *`offset`* must not be negative, and must be less than the buffer size, or a null pointer is returned an an appropriate error message is set; this message can be accessed by the `errMsg` function.

`int size() const;`

Return the size of the buffer.

`static const char* errMsg();`
Return the last error message (currently, only `previous()` sets error messages).

6.3 Class MultiPortHole

A MultiPortHole is an organized connection of related PortHoles. Any number of PortHoles can be created within the PortHole; their names have the form *`mphname#1`*, *`mphname#2`*, etc., where *`mphname`* is replaced by the name of the MultiPortHole. When a PortHole is added to the MultiPortHole, it is also added to the porthole list of the Block that contains the MultiPortHole. As a result, a Block that contains a MultiPortHole has, in effect, a configurable number of portholes. A pair of MultiPortHoles can be connected by a “bus connection”. This technique creates *`n`* PortHoles in each MultiPortHole and connects them all “in parallel”. The MultiPortHole constructor sets the “peer MPH” to 0. The destructor deletes any constituent PortHoles.

6.3.1 MultiPortHole public members

`void initialize();`
Does nothing.

`void busConnect (MultiPortHole& peer, int width, int delay = 0);`
Makes a bus connection with another multiporthole, *`peer,`* with width *`width`* and delay *`delay.`* If there is an existing bus connection, it is changed as necessary; an existing bus connection may be widened, or, if connected to a different peer, all constituent portholes are deleted and a bus is made from scratch.

`int isItMulti() const;`
Returns `TRUE`.

```
MultiPortHole& setPort(const char* portName,
                       Block* parent,DataType type = FLOAT);
int numberPorts() const;
```
Return the number of PortHoles in the MultiPortHole.

`virtual PortHole& newPort();`
Add a new physical port to the MultiPortHole list.

`MultiPortHole& realPort();`
Return the real MultiPortHole associated with me, translating any aliases.

`void setAlias (MultiPortHole &blockPort);`
Set my alias to *`blockPort.`*

`virtual PortHole& newConnection();`
Return a new port for connections. If there is an unconnected porthole, return the first one; otherwise make a new one.

6.3.2 MultiPortHole protected members

`PortList ports;`
The list of portholes (should be protected).

`const char* newName();`

This function generates names to be used for contained PortHoles. They are saved in the hash table provided by the `hashstring` function .

`PortHole& installPort(PortHole&` *`p`*`);`

This function adds a newly created port to the multiporthole. Derived MultiPortHole classes typically redefine `newPort` to create a porthole of the appropriate type, and then use this function to register it and install it.

`void delPorts();`

This function deletes all contained portholes.

6.4 AutoFork and AutoForkNode

AutoForks are a method for implementing netlist-style connections. An AutoForkNode is a type of Geodesic built on top of AutoFork. The classes are separate to allow a "mixin approach", so that if a domain requires special actions in its Geodesics, these special actions can be written only once and be implemented in both temporary and permanent connections. The implementation technique used is to automatically insert a Fork star to allow the n-way connection; this Fork star is created by invoking `KnownBlock::makeNew("Fork"),` which works only for domains that have a fork star.

6.4.1 Class AutoFork

An AutoFork object has an associated Geodesic and possibly an associated Fork star (which it creates and deletes as needed). It is normally used in a multiply inherited object, inherited from AutoFork and some kind of Geodesic; hence the associated Geodesic is the object itself. The constructor for class AutoFork takes a single argument, a reference to the Geodesic. It sets the pointer to the fork star to be null. The destructor removes the fork star, if one was created. There are two public member functions, `setSource` and `setDest.`

`PortHole* setSource(GenericPort&` *`port`*`, int` *`delay`* `= 0);`

If there is already an originating port for the geodesic, this method returns an error. Otherwise it connects it to the node.

`PortHole* setDest(GenericPort&` *`port`*`, int` *`alwaysFork`* `= 0);`

This function may be used to add any number of destinations to the port. Normally, when there is more than one output, a Fork star is created and inserted to support the multi-way connection, but if there is only one output, a direct connection is used. However, if *`alwaysFork`* is true, a Fork is inserted even for the first output. When the fork star is created, it is inserted in the block list for the parent galaxy (the parent of the geodesic).

6.4.2 Class AutoForkNode

Class AutoForkNode is multiply inherited from Geodesic and AutoFork. This class redefines `isItPersistent` to return `TRUE`, and redefines the `setSourcePort` and `setDestPort` functions to call the `setSource` and `setDest` functions of AutoFork. The exact same form could be used to generate other types of auto-forking nodes (that is, this class could have been done with a template).

6.5 Class ParticleStack

ParticleStack is an efficient base class for the implementation of structures that organize Particles. As Particles have a link field, ParticleStack is simply implemented as a linked list of Particles. Strictly speaking, a dequeue is implemented; particles can be inserted from either end. ParticleStack has some odd attributes; it is designed for very efficient implementation of Geodesic and Plasma to move around large numbers of Particle objects very efficiently.

`ParticleStack(Particle*` *h*);

The constructor takes a Particle pointer. If it is a null pointer an empty ParticleStack is created. Otherwise the stack has one particle. Adding a Particle to a ParticleStack modifies that Particle's link field; therefore a Particle can belong to only one ParticleStack at a time.

`~ParticleStack();`

The destructor deletes all Particles EXCEPT for the last one; we do not delete the last one because it is the "reference" particle (for Plasma) and is normally not dynamically created (this code may be moved in a future release to the Plasma destructor, as this behavior is needed for Plasma and not for other types of ParticleStack).

`void put(Particle*` *p*);

Push *p* onto the top (or head) of the ParticleStack.

`Particle* get();`

Pop the particle off the top (or head) of the ParticleStack.

`void putTail(Particle*` *p*);

Add *p* at the bottom (or tail) of the ParticleStack.

`int empty() const;`

Return `TRUE` (1) if the ParticleStack is empty, otherwise 0.

`int moreThanOne() const;`

Return `TRUE` (1) if the ParticleStack has two or more particles, otherwise 0. This is provided to speed up the derived class Plasma a bit.

`void freeup();`

Returns all Particles on the stack to their Plasma (the allocation pool for that particle type).

`Particle* head() const;`

Return pointer to head.

`Particle* tail() const;`

Return pointer to tail.

6.6 Class Geodesic

A Geodesic implements the connection between a pair, or a larger collection, of PortHoles. A Geodesic may be temporary, in which case it is deleted when the connection it implements is broken, or it can be permanent, in which case it can live in disconnected form. As a rule, temporary geodesics are used for point-to-point connections and permanent geodesics are used for netlist connections. In the latter case, the Geodesic has a name and is a member of a galaxy; hence, Geodesic is derived from NamedObj . The base class Geodesic, which is temporary, suffices for most simulation and code generation domains. In fact, in a number of these domains it contains unused features, so it is perhaps too “heavyweight” an object. A Geodesic contains a ParticleStack member which is used as a queue for movement of Particles between two portholes; it also has an originating port and a destination port. A Geodesic can be asked to have a specific number of initial particles. When initialized, it creates that number of particles in its ParticleStack; these particles are obtained from the Plasma of the originating port (so they will be of the correct type). A severe limitation of the current Geodesic class is that it is designed around point-to-point connections, ie, a single source port to a single destination port. This is a problem for domains that wish to support one-to-many geodesics (single source to multiple receivers) or many-to-many geodesics (such as multiple in/out ports connected to a common bus). Geodesic ought to be redesigned as a base class that supports any number of connected ports, with the restriction to point-to-point being a specialized subclass. This would also allow a cleaner treatment of autofork (autoforking geodesics could just be a subclass of Geodesic). It would be necessary to remove PortHole’s belief that there is a unique far-side porthole, and that would require rethinking the porthole type resolution algorithm; probably type resolution should become a Geodesic function, not a PortHole function. This area will be addressed in some future version of Ptolemy.

6.6.1 Geodesic public members

`virtual PortHole* setSourcePort (GenericPort &`*`src`*`, int `*`delay`*` = 0);`

Set the source port and the number of initial particles. The actual source port is determined by calling `newConnection` on *`src;`* thus if *`src`* is a MultiPortHole, the connection will be made to some port within that MultiPortHole, and aliases will be resolved. The return value is the “real porthole” used. In the default implementation, if there is already a destination port, any preexisting connection is broken and a new connection is completed.

`virtual PortHole* setDestPort (GenericPort &`*`dp`*`);`

Set the destination port to `dp.newConnection()`. The return value is the “real porthole” used. In the default implementation, if there is already a source port, any preexisting connection is broken and a new connection is completed.

`virtual int disconnect (PortHole & `*`p`*`);`

In the default implementation, if *`p`* is either the source port or the destination port, both the source port and destination port are set to null. This is not enough to break a connection; as a rule, `disconnect` should be called on the porthole, and that method will call this one as part of its work.

`virtual void setDelay (int `*`newDelay`*`);`

Modify the delay (number of initial tokens) of a connection. The default implementation simply changes a count.

`virtual int isItPersistent() const;`

Return `TRUE` if the Geodesic is persistent (may exist in a disconnected state) and `FALSE` otherwise. The default implementation returns `FALSE`.

`PortHole* sourcePort () const;`
`PortHole* destPort () const;`

Return my source and destination ports, respectively.

`virtual void initialize();`

In the default implementation, this function initializes the number of Particles to that given by the numInitialParticles field (the value returned by `numInit();` these Particles are obtained from the Plasma (allocation pool) for the source port. The particles will have zero value for numeric particles, and will hold the "empty message" for message Particles.

`void put(Particle*` *p*);

Put a particle into the Geodesic (using a FIFO discipline).

`Particle* get();`

Retrieve a particle from the Geodesic (using a FIFO discipline). Return a null pointer if the Geodesic is empty.

`void pushBack(Particle*` *p*);

Push a Particle back into the Geodesic (onto the front of the queue, instead of onto the back of the queue as `put` does).

`int size() const;`

Return the number of Particles on the Geodesic at the current time.

`int numInit() const;`

Return the number of initial particles. This call is valid at any time. Immediately after `initialize`, `size` and `numInit` return the same value (and this should be true for any derived Geodesic as well), but this will not be true during execution (where `numInit` stays the same and `size` changes).

`StringList print(int verbose = 0) const;`

Print information on the Geodesic, overrides NamedObj function.

`virtual void incCount(int);`
`virtual void decCount(int);`

These methods are available for schedulers such as the SDF scheduler to simulate a run and keep track of the number of particles on the geodesic. `incCount` increases the count, `decCount` decreases it, They are virtual to allow additional bookkeeping in derived classes.

`int maxNumParticles() const;`

Return maximum number of particles.

`virtual void makeLock(const PtGate& master);`

Create a lock for the Geodesic.

`virtual void delLock();`

Delete lock for the Geodesic.

```
int isLockEnabled() const;
```
Return lock status.

```
const char * initDelayValues();
```
Return the *initValues* string.

6.6.2 Geodesic protected members

```
void portHoleConnect();
```
This function completes a connection if the originating and destination ports are set up.

```
virtual Particle* slowGet();
virtual void slowPut(Particle*);
```
The “slow” versions of `get()` and `put()`.

```
PortHole *originatingPort;
PortHole *destinationPort;
```
These protected members point to my neighbors.

6.7 Class Plasma

Class Plasma is a pool for particles. It is derived from ParticleStack . Rather than allocating Particles as needed with `new` and freeing them with `delete,` we instead provide an allocation pool for each type of particle, so that very little dynamic memory allocation activity will take place during simulation runs. All Plasma objects known to the system are linked together. As a rule, there is one Plasma for each type of particle; however, each of these objects is of type Plasma, not a derived type. At all times, a Plasma has at least one Particle in it; that Particle’s virtual functions are used to clone other particles as needed, determine the type, etc. The constructor takes one argument, a reference to a Particle. It creates a one-element ParticleStack, and links the Plasma into a linked list of all Plasma objects. The `put` function (for putting a particle into the Plasma) adds a particle to the Plasma’s ParticleStack. As a rule, it should not be used directly; the Particle’s `die` method will automatically add it to the right Plasma (future releases may protect this method to prevent its general use).

```
Particle* get();
```
This function gets a Particle from the Plasma, creating a new one if the Plasma has only one Particle on it (we never give away the last Particle).

```
int isLocal() const;
```
Returns *localFlag*.

```
static Plasma* getPlasma (DataType t);
```
Get the appropriate global Plasma object given a type.

```
static Plasma* makeNew (DataType t);
```
Create a local Plasma object given a type.

```
void makeLock(const PtGate& master);
```
Create a lock for the Plasma.

`void delLock();`

Delete lock for the Plasma. No effect on global plasmas.

`short incCount();`

Increase reference count, when adding reference from PortHole to a local Plasma. New count is returned. Global Plasmas pretend their count is always 1.

`short decCount();`

Decrease reference count, when removing reference from PortHole to a local Plasma. New count is returned. Idea is we can delete it if it drops to zero. Global Plasmas pretend their count is always 1.

`DataType type();`

Returns the type of the particles on the list (obtained by asking the head Particle).

`static Plasma* getPlasma(DataType` *`type`*`);`

Searches the list of Plasmas for one whose type matches the argument, and returns a pointer to it. A null pointer is returned if there is no match.

6.8 Class ParticleQueue

Class ParticleQueue implements a queue of Particles. It uses a member of class ParticleStack to store the particles; it is not implemented by deriving from ParticleStack. It can implement a queue with finite or unlimited capacity. Rather than placing user-supplied Particles on the queue and removing them directly, it takes over the responsibility for memory management by allocating its own Particles from the Plasma and returning them as needed. When a user puts a Particle into the queue, the value of the Particle is copied (with the Particle `clone` method); similarly, when a user gets a Particle from the queue, he or she supplies a Particle to received the copied value. The advantage of this is that the user need not worry about lifetimes of Particles – when to create them, when it is safe to return them to the Plasma or delete them. The ParticleQueue default constructor forms an empty, unlimited capacity queue. There is also a constructor of the form

`ParticleQueue(unsigned int` *`cap`*`);`

This creates a queue that can hold at most *`cap`* particles. The destructor returns all Particles in the queue to their Plasma.

`int empty() const;`

Return `TRUE` if the queue is empty, else `FALSE`.

`int full() const;`

Return `TRUE` if the length equals the capacity, else `FALSE`.

`unsigned int capacity() const;`

Return the queue's capacity. If unlimited, the largest possible unsigned int on the machine will be returned.

`unsigned int length() const;`

Return the number of particles in the queue.

`int putq(Particle& p);`

Put a copy of particle *p* into the queue, if there is room. Returns `TRUE` on success, `FALSE` if the queue is already at capacity.

`int getq(Particle& p);`

Get a particle from the queue, and copy it into the user-supplied particle *p*. This returns `TRUE` on success, `FALSE` (and *p* is unaltered) if the queue is empty.

`void setCapacity(int sz);`

Modify the capacity to *sz,* if *sz* is positive or zero. If negative, the capacity becomes infinite.

`void initialize();`

Free up the queue contents. Particles are returned to their pools and the queue becomes empty.

`void initialize(int n);`

Equivalent to `initialize()` followed by `setCapacity(n)`.

6.9 Classes for Galaxy ports

Class GalPort is derived from class PortHole . Class GalMultiPort is derived from class MultiPortHole . These classes are used by InterpGalaxy , and in other places, to create galaxy ports and multiports that are aliased to some port of a member block. The constructor for each of these classes takes one argument, the interior port that is to be the alias. The `isItInput()` and `isItOutput()` functions are implemented by forwarding the request to the alias.

6.10 The PortHole type resolution algorithm

The type resolution algorithm is concerned with assigning concrete types to `ANYTYPE` portholes and resolving conflicts between the types of connected portholes. The algorithm is somewhat complex since it tries to produce convenient results in an area where the "right" behavior is not always easy to define. Ptolemy 0.7 introduces a new resolution algorithm that is hoped to produce more convenient and less surprising results than the method previously used. The problem of connecting ports of different types is simple to resolve: we say that the input porthole determines what particle type to use, and that any necessary type conversion takes place when the output porthole puts data into a particle (or buffer variable, in the case of codegen domains). The opposite convention would be about equally defensible, but this is the one that has historically been used in Ptolemy. It has the advantage that star writers can presume that the declared type of an input porthole is the data type that will actually be received, whenever the declared type is not `ANYTYPE`. Resolving `ANYTYPE` portholes is much more difficult. We need to handle several fundamental cases:

1. Printer and similar polymorphic stars, which accept any input type. They simply declare their inputs to be `ANYTYPE`, and we need to resolve them to the type of the connected output. (Introducing any forced particle type conversion would be very

undesirable.)

2. Fork and similar stars, which want to bind multiple outputs to the type of a given input. Here the input porthole and output portholes are `ANYTYPE`, and are bound into a type equivalence set by `inheritTypeFrom()`. If possible we want to resolve all these portholes to the type of the output porthole connected to the input.

3. Merge and similar stars, which have a single output type-equivalenced to multiple inputs. If the inputs all receive the same type of data, we should resolve the output to that type. If there is no common input type, but the input connected to the single output has determinable type, we can resolve the output porthole to that type (since the data would ultimately get converted to that type, anyway). If the connected input is `ANYTYPE`, we must declare error, because we have no good way to choose a type for the Merge's output.

We have to recursively propagate type information in order to deal with chains of `ANYTYPE` stars, such as one Fork following another. In some cases the type is really undefined. Consider this universe (using ptcl syntax):

```
star f Fork; star p Printer
connect f output f input 1
connect f output p input
```

There are no types anywhere in the system. We have little choice but to declare an error. Thus, the fact that we will sometimes fail to assign a type is not an implementation shortcoming but an unavoidable property of the problem. These considerations lead to the following algorithm. We first perform a recursive scan to resolve `ANYTYPE` portholes, the results of which are represented by a "preferred" type assigned to each porthole. (A porthole of non-`ANYTYPE` declared type always has that type as its preferred type; so preferred type assignment is only interesting for `ANYTYPE` portholes.) Then, the "resolved" type of each connected pair of input and output portholes is the preferred type of the input porthole. This is the type that will be used for actual data transported between those portholes. It is useful to explicitly store the preferred type, so that codegen domains can detect type mismatches just by looking at individual output portholes. A type conversion star can be spliced in wherever an output is found that has `preferredType != resolvedType`. Assignment of preferred types proceeds in two passes. Pass 1 is "feed forward" from outputs to inputs. Pass 2 is "feed back" from inputs to outputs; it is the dual of Pass 1. Pass 2 is invoked only if Pass 1 is unable to choose a type for a porthole. The details are:

Pass 1:

Non-`ANYTYPE` portholes are simply assigned their declared type as preferred type. If an `ANYTYPE` porthole is not a member of an equivalence group, but is an input porthole and is connected to a porthole of pass-1-assignable type, that porthole's type becomes its preferred type. When an `ANYTYPE` porthole is a member of an equivalence group, the group is scanned to see if it includes any non-`ANYTYPE` portholes; if so, they must all agree in type, and that type becomes the preferred type of all members of the group. But usually, all the members of an equivalence set will be `ANYTYPE`. Then, pass 1 scans all the input portholes of the group to see whether their connected portholes have pass-1-assignable type. If at least one does, and all of the assignable ones have the same preferred type, then that common type becomes the preferred type of all the members of the equivalence group.

Pass 2:

If an unassigned `ANYTYPE` porthole is not a member of an equivalence group, but is connected to a porthole of type assignable by either pass 1 or pass 2, that porthole's type becomes its preferred type. When an `ANYTYPE` porthole is a member of an equivalence group, all the output portholes of the group are scanned to see whether their connected portholes have type assignable by either pass 1 or pass 2. If at least one does, and all of the assignable ones have the same preferred type, then that common type becomes the preferred type of all the members of the equivalence group.

Pass 1 handles Fork-like stars as well as Merge stars whose inputs all have the same type. Pass 2 does something reasonable with Merge stars that have inputs of different types: if the merge output is going to a port of knowable type, we may as well just output particles of that type. An error is declared if a porthole's type remains unassigned after both passes. This occurs if a Merge-like star has inputs of nonidentical types and an output connected to an (unresolvable) `ANYTYPE` input. The user must insert type conversion stars to coerce the Merge inputs to a common type, so that the system can figure out what type to use for the Merge output. Notice that each pass will resolve an equivalence set if all the **assignable** connected portholes agree on a type; it is not required that all the connected portholes be assignable. This rule is needed to allow resolution of schematics that contain type-free loops. Here is an example:

```
star imp Impulse; star f Fork; star d Delay; star p Printer
connect imp output f input#1
connect f output d input
connect d output f input#2
connect f output p input
```

This schematic will work if we resolve all the ports to `FLOAT` (the output type of Impulse). But if we insist on both Fork inputs being resolved before we assign a type to the Fork output, we will be unable to resolve the schematic. So, once Pass 1 has recursively traversed the schematic and concluded that it can't yet assign a type to Fork's `input#2`, it uses the `FLOAT` type found at `input#1` to resolve the type of the Fork portholes. Further recursion then propagates this result around the schematic. This rather baroque-looking algorithm does have some simple properties. In particular, all members of an equivalence set are guaranteed to be assigned the same preferred type; an error will be reported if this is not possible. In some domains it is important that members of an equivalence set have the same resolved type, not just the same preferred type. (For example, the CGC Fork star fails if this is not so, because its various portholes are all just aliases for a single variable.) The domain can check this by seeing whether preferred type equals resolved type for all portholes. If the types are not the same, it can either report an error or splice in a type-conversion star to make them the same.

Note:

It might be better to cause this to happen on a per-star-type basis, not a per-domain basis, since one can imagine that some CG blocks would need strict type equality of portholes while others would not. This improvement is not currently implemented. The porthole type resolution algorithm is dependent on the notion that every porthole is connected to just one other porthole. If class Geodesic is ever redesigned to support multiple connections directly, some work would be needed. A likely tactic is to move some or all of the resolution work into Geodesic. A one-to-one Geodesic could enforce the same behavior described above, but one-to-many or many-to-many Geodesics would need different, possibly domain-dependent behavior.

6.11 Changes since Ptolemy0.6

Ptolemy 0.7 introduces several changes related to porthole type resolution. Older versions used a much simpler `ANYTYPE` resolution algorithm, which essentially amounted to just pass 2 ("feed back") of the present method. That had the serious deficiency that it couldn't decide what to do with fork stars feeding inputs of multiple types. For example, a fork star receiving `INT` and outputting to `INT` and `FLOAT` portholes would lead to a type resolution error. In essence, the old code insisted on being able to push type conversions back across a fork, and would fail if it couldn't assign the same type to all the fork outputs. The new algorithm solves this problem by delaying type conversions until after a fork. Occasionally this will introduce some inefficiency. For example, if a fork receives `INT` and feeds two `FLOAT`s, the new method leads to a type conversion being done separately on each fork output, whereas the old method would have generated only one conversion at the fork input. The improved ease of use of the new method is judged well worth this loss.

Formerly, the member ports of a multiporthole were always constrained to have the same resolved type. This is no longer true, since it gets in the way for polymorphic stars. But if a multiporthole is tied to another porthole via `inheritTypeFrom`, then each member porthole will still be constrained to match the type of that other porthole, at least in terms of preferred type. Formerly, the HOF stars acted during the galaxy "setup" phase, in which each block within the galaxy receives its setup call. This proved inadequate because porthole type resolution is done during setup; by the time a HOF star acted, the types of the portholes connected to it would already have been resolved. For example, a HOFNop star formerly constrained all the particles passing through it to be of the same type, because the porthole resolver insisted on being able to choose a unique type for the HOFNop's output porthole, even though that porthole is just a dummy that won't even exist at runtime. In Ptolemy 0.7, a "preinitialize" phase has been added so that HOF stars can rewire the galaxy and remove themselves before any block setup or porthole type resolution occurs. The constraints on porthole types are then only those resulting from the rewired schematic.

Chapter 7. Particles and Messages

Authors: *Joseph T. Buck*

7.1 Class Particle

A Particle is a little package that contains data; they represent the principal communication technique that blocks use to pass results around. They move through PortHoles and Geodesics; they are allocated in pools called Plasmas. The class Particle is an abstract base class; all real Particle objects are really of some derived type. All Particles contain a link field that allows queues and stacks of Particles to be manipulated efficiently (class ParticleStack is a base class for everything that does this). Particles also contain virtual operators for loading and accessing the data in various forms; these functions permit automatic type conversion to be easily performed.

7.2 Particle public members

```
virtual DataType type() const = 0;
```

Return the type of the particle. DataType is actually just a typedef for `const` char*, but when we use DataType, we treat it as an abstract type. Furthermore, two DataType values are considered the same if they compare equal, which means that we must assure that the same string is always used to represent a given type.

```
virtual operator int () const = 0;
virtual operator float () const = 0;
virtual operator double () const = 0;
virtual operator Complex () const = 0;
```

These are the virtual casting functions, which convert the data in the Particle into the desired form. The arithmetic Particles support all these functions cleanly. Message particles may return errors for some of these functions (they must return a value, but may also call `Error::abortRun.`

```
virtual StringList print () const = 0;
```

Return a printable representation of the Particle's data.

```
virtual void initialize() = 0;
```

This function zeros the Particle (where this makes sense), or initializes it to some default value.

```
virtual void operator << (int arg) = 0;
virtual void operator << (double arg) = 0;
virtual void operator << (const Complex& arg) = 0;
```

These functions are, in a sense, the inverses of the virtual casting operators. They load the particle with data from *`arg,`* performing the appropriate type conversion.

```
virtual Particle& operator = (const Particle& arg) = 0;
```

Copy a Particle. In general, we permit this only for Particles of the same type, and otherwise assert an error. But the arithmetic particle types invoke type conversion, via the virtual casting operators, so as to allow assignment from other arithmetic particle types. Without this exception, useful cases such as forking an INT output to INT and FLOAT inputs would fail in the simulation domains (because the fork stars use particle assignment).

```
virtual int operator == (const Particle&) = 0;
```

Compare two particles. As a rule, Particles will be equal only if they have the same type, and, in a sense that is separately determined for each type, the same value.

```
virtual Particle* clone() const = 0;
```

Produce a second, identical particle (as a rule, one is obtained from the Plasma for the particle if possible).

```
virtual Particle* useNew() const = 0;
```

This is similar to `clone,` except that the particle is allocated from the heap rather than from the Plasma.

```
virtual void die() = 0;
```

Return the Particle to its Plasma.

```
virtual void getMessage (Envelope&);
virtual void accessMessage (Envelope&) const;
virtual void operator << (const Envelope&);
```

These functions are used to implement the Message interface. The default implementation returns errors for them; it is only if the Particle is really a MessageParticle that they successfully send or receive a Message from the Particle.

7.3 Arithmetic Particle classes

There are three standard arithmetic Particle classes: IntParticle, FloatParticle, and ComplexParticle. As their names suggest, each class adds to Particle a private data member of type int, double (not float!), and class Complex, respectively. When a casting operator or "<<" operator is used on a particle of one of these types, a type conversion may take place. If the type of the argument of cast matches the type of the particle's data, the data is simply copied. If the requested operation involves a "widening" conversion (int to float, double, or Complex; float to double or Complex; double to Complex), the "obvious" thing happens. Conversion from double to int rounds to the nearest integer; conversion from Complex to double returns the absolute value (not the real part!), and Complex to int returns the absolute value, rounded to the nearest integer. `initialize` for each of these classes sets the data value to zero (for the appropriate domain). The DataTypes returned by these Particle types are the global symbols INT, FLOAT, and COMPLEX, respectively. They have the string values "INT", "FLOAT", and "COMPLEX".

7.4 The Heterogeneous Message Interface

The heterogeneous message interface is a mechanism to permit messages of arbitrary type (objects of some derived type of class Message) to be transmitted by blocks. Because these messages may be very large, facilities are provided to permit many references to the same Message; Message objects are "held" in another class called Envelope. As the name suggests, Messages are transferred in Envelopes. When Envelopes are copied, both Envelopes refer to the same Message. A Message will be deleted when the last reference to it disappears; this means that Messages must always be on the heap. So that Messages may be transmitted by portholes, there is a class MessageParticle whose data field is an Envelope. This permits it to hold a Message just like any other Envelope object.

7.4.1 Class Envelope

class Envelope has two constructors. The default constructor constructs an "empty" Envelope (in reality, the envelope is not empty but contains a special "dummy message" – more on this later). There is also a constructor of the form

`Envelope(Message&` *`data`*`);`

This constructor creates an Envelope that contains the Message *`data,`* which MUST have been allocated with `new.` Message objects have reference counts; at any time, the reference count equals the number of Envelope objects that contain (refer to) the Message object. When the reference count drops to zero (because of execution of a destructor or assignment operator on an Envelope object), the Message will be deleted. Class Envelope defines an assignment operator, copy constructor, and destructor. The main work of these functions is to manipulate reference counts. When one Envelope is copied to another, both Envelopes refer to the same message.

`int empty() const;`

Return TRUE if the Envelope is "empty" (points to the dummy message), FALSE otherwise.

`const Message* myData() const;`

Return a pointer to the contained Message. This pointer must not be used to modify the Message object, since other Envelopes may refer to the same message.

`Message* writableCopy();`

This method produces a writable copy of the contained Message, and also zeros the Envelope (sets it to the empty message). If this Envelope is the only Envelope that refers to the message, the return value is simply the contained message. If there are multiple references to the message, the `clone` method is called on the Message, making a duplicate, and the duplicate is returned. The user is now responsible for memory management of the resulting Message. If it is put into another Envelope, that Envelope will take over the responsibility, deleting the message when there is no more need for it. If it is not put into another Envelope, the user must make sure it is deleted somehow, or else there will be a memory leak.

`int typeCheck(const char*` *`type`*`) const;`

This member function asks the question "is the contained Message of class *`type,`* or derived from *`type"`* ? It is implemented by calling `isA` on the Message. Either TRUE or FALSE is returned.

`const char* typeError(const char* *expected*) const;`

This member function may be used to format error messages for when one type of Message was expected and another was received. The return value points to a static buffer that is wiped out by subsequent calls.

```
const char* dataType() const;
int asInt() const;
double asFloat() const;
Complex asComplex() const;
StringList print() const;
```

All these methods are “passthrough methods”; the return value is the result of calling the identically named function on the contained Message object.

7.4.2 Class Message

Message objects can be used to carry data between blocks. Unlike Particles, which must all be of the same type on a given connection, connections that pass Message objects may mix message objects of many types on a given connection. The tradeoff is that blocks that receive Message objects must, as a rule, type-check the received objects. The base class for all messages, named Message, contains no data, only a reference count (accordingly, all derived classes have a reference count and a standard interface). The reference count counts how many Envelope objects refer to the same Message object. The constructor for Message creates a reference count that lives on the heap. This means that the reference count is non-const even when the Message object itself is const. The copy constructor for Message ignores its argument and creates a new Message with a new reference count. This is necessary so that no two messages will share the same reference count. The destructor, which is virtual, deletes the reference count. The following Message functions must be overridden appropriately in any derived class:

`virtual const char* dataType() const;`

This function returns the type of the Message. The default implementation returns “DUMMY”.

`virtual Message* clone() const;`

This function produces a duplicate of the object it is called on. The duplicate must be “good enough” so that applications work the same way whether the original Message or one produced by `clone()` is received. A typical strategy is to define the copy constructor for each derived Message class and write something like

```
Message* MyMessage::clone() const { return new MyMessage(*this);}
virtual int isA(const char*) const;
```

The `isA` function returns true if given the name of the class or the name of any base class. Exception: the base class function returns FALSE to everything (as it has no data at all). A macro `ISA_FUNC` is defined to automate the generation of implementations of derived class `isA` functions; it is the same one as that used for the NamedObj class. The following methods may optionally be redefined.

`virtual StringList print() const;`

This method returns a printable representation of the Message. The default implementation returns a message like

`Message class <`*`type`*`>`: no print method

where *`type`* is the message type as returned by the `dataType` function.

```
virtual int asInt() const;
virtual double asFloat() const;
virtual Complex asComplex() const;
```

These functions represent conversions of the Message data to an integer, a floating point value, and a complex number, respectively. Usually such conversions do not make sense; accordingly, the default implementations generate an error message (using the protected member function `errorConvert)` and return a zero of the appropriate type. If a conversion does make sense, they may be overridden by a method that does the appropriate conversion. These methods will be used by the MessageParticle class when an attempt is made to read a MessageParticle in a numeric context. One protected member function is provided:

`int errorConvert(const char*` *`cvttype`*`)` const;

This function invokes `Error::abortRun` with a message of the form

`Message class <`*`msgtype`*`>`: invalid conversion to *`cvttype`*

where *`msgtype`* is the type of the Message, and *`cvttype`* is the argument.

7.4.3 Class MessageParticle

MessageParticle is a derived type of Particle whose data field is an Envelope; accordingly, it can transport Message objects. MessageParticle defines no new methods of its own; it only provides behaviors for the virtual functions defined in class Particle. The most important such behaviors are as follows:

`void operator << (const Envelope&` *`env`*`)`;

This method loads the Message contained in *`env`* into the Envelope contained in the MessageParticle. Since the Envelope assignment operator is used, after execution of this method both *`env`* and the MessageParticle refer to the message, so its reference count is at least 2.

`void getMessage(const Envelope&` *`env`*`)`;

This method loads the message contained in the MessageParticle into the Envelope *`env,`* and removes the message from the MessageParticle (so that it now contains the dummy message). If *`env`* previously contained the only reference to some other Message, that previously contained Message will be deleted.

`void accessMessage(const Envelope&` *`env`*`)`;

`accessMessage` is the same as `getMessage` except that the message is not removed from the MessageParticle. It can be used in situations where the same Particle will be read again. We recommend that `getMessage` be used where possible, especially for very large message objects, so that they are deleted as soon as possible.

7.5 Example Message types

The kernel provides two simple sample message types for transferring arrays of data. They are almost identical except that one holds an array of integers and the other holds an array of single precision floating point data. The array contents live on the heap. Each is derived from class Message. Each provides a public data member that points to the data. As a rule, we recommend against public data members for classes, but an exception was made in this case, perhaps unwisely. This section will describe the interface of the FloatVecData class. The interface for IntVecData is almost identical. Three constructors are provided:

`FloatVecData(int` *`len`*`);`

This form creates an uninitialized array of length *`len`* in the FloatVecData object. Since the pointer to the data is public the array may easily be filled in.

`FloatVecData(int` *`len`*`,const float *`*`srcData`*`);`

This form creates an array of length *`len`* and initializes it with *`len`* elements from *`srcData.`*

`FloatVecData(int` *`len`*`,const double *`*`srcData`*`);`

This form is the same, except that the source data is double precision (it is converted to single precision). This is the only function for which an analogous function does not exist in IntVecData (an IntVecData can only be initialized from an integer array). An appropriate copy constructor, assignment operator, and destructor are defined.

`int length() const;`

Return the length of the array.

`float *data;`

Public data member; points to the array. It is permissible to read or assign the *`len`* elements starting at `data;` the effect of altering the `data` pointer itself is undefined.

`const char* dataType() const;`

Returns the string `"FloatVecData".`

`int isA(const char*` *`type`*`) const;`

TRUE for *`type`* equal to `"FloatVecData",` otherwise false.

`StringList print() const;`

Returns a comma-separated list of elements enclosed in curly braces.

`Message* clone() const;`

Creates an identical copy with `new.`

Chapter 8. The incremental linker

Authors: *Joseph T. Buck*
Christopher Hylands

The incremental linker permits user written code to be added to the system at runtime. Two different mechanisms are provided, called a temporary link and a permanent link. With either a temporary link or a permanent link, code is linked using the incremental linking facilities of the Unix linker, the new code is read into the Ptolemy executable, and symbols corresponding to C++ global constructors are located and called. This means that such code is expected to register objects on Ptolemy's known lists (e.g. KnownBlock, KnownState, or KnownTarget) so that new classes become usable. *Warning:* if the executable containing the Linker class is stripped, the incremental linker will not work!

8.1 ld -A style linking vs. dlopen() style linking

There are two ways incremental linking is implemented: "ld -A" and "dlopen()" style linking.

The first type of implementation uses a BSD Sun-style loader with the -A flag to load in .o files. Usually, binaries that are to be dynamically linked must be built with the -N option. This is the older style of linking present in Ptolemy0.5 and earlier.

The second type of implementation uses the System V Release 4 `dlopen()` call to load in shared objects (.so files). SunOS4.1.x, Solaris2.x and Irix5.x support this style of dynamic linking. In Ptolemy0.6, only the sol2, sol2.cfront, and hppa architectures support dynamic linking of shared objects.

The interface to both styles of linking is very similar, though there are differences.

8.2 Temporary vs. Permanent Incremental Linking

Code that is linked in by the "temporary link" technique does not alter the symbol table in use. For that reason, subsequent incremental links, whether temporary or permanent, cannot "see" any code that was linked in by previous temporary links. The advantage is that the same symbols (for example, a Ptolemy star definition) may be redefined, which is useful in code development, as buggy star definitions can be replaced by valid ones without exiting Ptolemy. Code that is linked in by the "permanent link" method has the same status as code that was linked into the original executable. With "ld -A" style incremental linking, a permanent link creates or replaces the `.pt_symtable` file in the directory in which Ptolemy was started. This file contains the current symbol table for use by subsequent links, temporary or permanent. This file is deleted when the Ptolemy process exits normally. It is left around when the process crashes, as it is useful for debugging (as it contains symbols for object files that were incrementally linked using the permanent method as well as those in the original executable). With `dlopen()` style incremental linking, we keep track of all the files that have been permanently linked in. After a file has been permanently linked in, each successive link (permanent or not) includes all the permanently linking in files. That is, if we permanently link in foo.o, then when we link in bar.o, we will generate a shared object file that includes both foo.o and bar.o, and then call `dlopen()` on that shared object file. Note that with `dlopen()` style linking it is possible to relink in stars that have been permanently linked. When a file is to be linked in, we check against the list of permanently linked in file names and remove any duplicates. This method of checking will fail if one permanently links in a file, and then links in the same file as a different name, perhaps through the use of symbolic links. That is, if we permanently link in `./foo.o` and `./bar.o` is a link to `./foo.o,` when we link in `./bar.o,` we will have multiple symbols defined, and we will get an error. In other words, we only check for duplicate file names, we do not check for duplicate symbols in any files, the loader does this for us. Currently each `dlopen()` style link generates a temporary shared object in `/tmp.` If you are doing a large number of `dlopen()` style permanent links, you will have many files in `/tmp.` We hope to resolve this potential problem in a later release. Eventually, it would be nice if the code read the value of an optional environment variable, such as `TMPDIR.`

8.3 Linker public members

`static void init(const char*` *`execName`*);

This function initializes the linker module by telling it where the executable for this program is. For most purposes, passing it the value of `argv[0]` passed to the `main` function will suffice.

`static int linkObj(const char*` *`objName`*);

Link in a single object module using the temporary link mechanism (this entry point is provided for backward compatibility).

`static int multiLink(const char*` *`args`*, int *`permanent`*);
`static int multiLink(int` *`argc`*, char** *`argv`*);

Both of these functions give access to the main function for doing an incremental link. They permit either a temporary or a permanent link of multiple files; flags to the Unix linker such as `-l` to specify a library or `-L` to specify a search directory for libraries are permitted. For the first form, *`args`* are passed as part of a linker command that is expanded by the Unix shell. A permanent link is performed if *`permanent`* is true (nonzero); otherwise a temporary link is performed. The second form is provided for ease of interfacing to the Tcl interpreter, which likes to pass arguments to commands in this style. In this case, *`argv[0]`* indicates the type of link: if it begins with the character `p,` a permanent link is performed; otherwise a temporary link is performed. The remaining arguments are concatenated (separated by spaces) and appear in the argument to the Unix linker.

`static int isActive();`

This function returns TRUE if the linker is currently active (so objects can be marked as dynamically linked by the known list classes). Actually the flag it returns is set while constructors or other functions that have just been linked are being run.

`static int enabled();`

Returns true if the linker is enabled (it is enabled by calling `Linker::init` if that function returns successfully). On platforms that do not support dynamic linking, this function always returns false (zero).

`static const char* imageFileName();`
Return the fully-expanded name of the executable image file (set by `Linker::init).`

`static void setDefaultOpts(const char*` *newValue*);
`static const char* defaultOpts();`
These functions set or return the linker’s default options, a set of flags appended to the end of the command line by all links.

8.4 Linker implementation

For each port of Ptolemy to a particular release, the Linker is implemented in one of two styles: “ld -A” style or “dlopen()” style. We discuss each style below.

8.4.1 Shared Objects and dlopen() style linking

If a Ptolemy release on a platform supports `dlopen` style dynamic linking, then the ptcl `link` command can be called with either a `.o` file or a `.so` file. If the `link` ptcl command is passed a `.o` file, then a `.so` file will be generated. If link ptcl command is passed a `.so` file, then the `.so` file will be loaded. If the `.so` file does not exist, then an error message will be produced and the link will return. There are several ways to specify the path to a shared object.

1. Using just a file name `link foo.so` will not work unless LD_LIBRARY_PATH includes the directory where `foo.so` resides. The man pages for `dlopen()` and `ld` discuss LD_LIBRARY_PATH Interestingly, using `putenv()` to set LD_LIBRARY_PATH from within ptcl has no effect on the runtime loader.
2. 2If the file name begins with `./`, then the current directory is searched. `link ./foo.so` should work, as will `link ./mydir/foo.so`.
3. If the file name is an absolute path name, then the shared object will be loaded. `link /tmp/foo.so` should work.
4. Dynamic programs can have a run path specified at link time. The run path is the path searched at runtime for shared object. (Under Solaris2.3, the `-R` option to `ld` controls the run path. Under Irix5.2, the `-rpath` option to `ld` controls the run path). If ptcl or pigiRpc has been compiled with a run path built in, and the shared object is in that path, then the shared object will be found. The Sun Linker Manual says: “To locate the shared object foo.so.1, the runtime linker will use any LD_LIBRARY_PATH definition presently in effect, followed by any runpath specified during the link-edit of prog and finally, the default location /usr/lib. If the file name had been specified ./foo.so.1, then the runtime linker would have searched for the file only in the present working directory.”

8.4.2 Porting the Dynamic Linking capability

This section is intended to assist those that attempt to port the Linker module to other platforms. The Linker class is implemented in three files: `Linker.h,` specifying the class interface, `Linker.cc,` specifying the implementation, and `Linker.sysdep.h,` specifying all the machine dependent parts of the implementation. To turn on debugging, compile `Linker.cc` with the DEBUG flag defined. One way to do this would be:

```
cd $PTOLEMY/obj.$PTARCH/kernel; rm -f Linker.o; make OPTIMIZER=-DDEBUG
```

The Linker class currently uses "ld -A" style dynamic linking on the Sun4 (Sparc) running SunOS4.1 and `g++`, the Sun4 (Sparc) running SunOS4.1 and Sun's `cfront` port, DECStations running Ultrix, HP-PA running `g++` or HP's `cfront` port. The Linker class currently uses "dlopen()" style dynamic linking on the Sun4 (Sparc) running Solaris2.4 and `g++`, the Sun4 (Sparc) running Solaris2 and Sun's `cfront` port (Sun `CC-3.0`), the Sun4 (Sparc) running Solaris2 and Sun's native C++ compiler `CC-4.0`, and SGI Indigos running IRIX-5.2 and `g++`. The intent is to structure the code in such a way that no `#ifdefs` appear in `Linker.cc`; they should all be in `Linker.sysdep.h`.

8.4.3 ld -A Style Dynamic Linking

The linker reads all new code into a pre-existing large array, rather than creating blocks of the right size with `new`, because the right size is not known in advance but a starting location must, as a rule, be passed to the loader in advance. This means that there is a wired-in limit to how much code can be linked in. The symbol `LINK_MEMORY`, which is set to one megabyte by default, is easily changed if required. Here are the steps taken by the linker to do its work:

1. Align the memory as required.
2. Form the command line and execute the Unix linker. Only certain flags in the command line will be system-dependent.
3. Read in the object file. This is heavily system-dependent.
4. Make the read-in text executable. On most systems this is a do-nothing step. On some platforms (such as HP) it is necessary to flush the instruction cache and that would be done at this point.
5. Invoke constructors in the newly read in code. Constructors are found by use of the `nm` program; the output is parsed to search for constructor symbols, whose form depends on the compiler used.
6. If this is a permanent link, copy the linker output to file `.pt_symtable`; otherwise delete it.

8.4.4 dlopen() Style Dynamic Linking

Here's how we link in an object using `dlopen()` style linking.

1. Generate a list of files to be linked in. If we have not yet done a permanent link, then the list of files to be linked in will consist of only the files in this link or multilink command. If the link is a permanent link, then we save the object name. For each successive link, we check the name of the object to be linked in against the list of objects permanently linked for duplicate file names. For each link after a permanent link, we include the names of all the unique permanently linked in objects in the generation of a temporary shared object file.

2. Generate a shared object `.so` file from all the objects to be linked in. The `.so` file is created in /tmp.
3. Do a `dlopen()` on the shared object.
4. Most architectures use `nm` to search for constructors, which are then invoked. Currently, sol2.cfront does not need to search for, or invoke constructors. gcc-2.5.8 has patches that allow similar functionality, but apparently these patches are not in gcc-2.6.0. Shared libraries in the SVR4 implementation contain optional `__init` and `__fini` functions, called when the library is first connected to (at startup or `dlopen()`) and when the library is disconnected from (at `dlclose()` or program exit), respectively. Some C++ implementations can arrange for these `__init` and `__fini` functions to contain calls to all the global constructors or destructors. On platforms where this happens, such as sol2.cfront, there is no need for the Linker class to explicitly call the constructors, as this will happen automatically.

Chapter 9. Parameters and States

Authors: *Joseph T. Buck*

Other Contributors: *Neil Smyth*

A State is a data structure associated with a block, used to remember data values from one invocation to the next. For example, the gain of an automatic gain controller is a state. A state need not be dynamic; for instance, the gain of fixed amplifier is a state. A parameter is the initial value of a state. A State actually has two values: the initial value, which is always a character string, and a current value, whose type is different for each derived class of State: integer for IntState, an array of real values for FloatArrayState, etc. In addition, states have attributes, which represent logical properties the state either has or does not have.

9.1 Class State

Class State is derived from class NamedObj. The State base class is an abstract class; you cannot create a plain State. The base class contains the initial value, which is always a `const char*`; the derived classes are expected to provide current values of appropriate type. The constructor for class State sets the initial value to a null pointer, and sets the state's attributes to a value determined by the constant AB_DEFAULT, which is defined in "State.h" to be the bitwise or of AB_CONST and AB_SETTABLE. The destructor does nothing extra.

9.1.1 State public members

```
State& setState(const char* stateName, Block* parent,
               const char* initValue, const char* desc = NULL);
```

This function sets the name, parent, initial value, and optionally the descriptor for a state. The character strings representing the initial value and descriptor must outlive the State.

```
State& setState(const char* stateName, Block* parent,
               const char* initValue, const char* desc,
               Attribute attr);
```

This function is the same as the other `setState,` but it also sets attributes for the state. The Attribute object represents a set of attribute bits to turn on or off.

```
void setInitValue(const char* valueString);
```

This function sets the initial value to *`valueString.`* This string must outlive the State.

```
const char* initValue () const;
```

Return the initial value.

```
virtual const char* type() const = 0;
```

Return the type name (for use in user interfaces, for example). When states are created dynamically (by the `KnownState` or `InterpGalaxy` class), it is this name that is used to specify the type.

`virtual int size() const;`

Return the size (number of distinct values) in the state. The default implementation returns 1. Array state types will return the number of elements.

`virtual int isArray() const;`

Return TRUE if this state is an array, false otherwise. The default implementation returns false.

`virtual void initialize() = 0;`

Initialize the state. The `initialize` function for a state is responsible for parsing the initial value string and setting the current value appropriately; errors are signaled using the `Error::abortRun` mechanism.

`virtual StringList currentValue() const = 0;`

Return a string representation of the current value.

`void setCurrentValue(const char*` *`newval`*);

Modify the current value, in a type-independent way. Notice that this function is not virtual. It exploits the semantics of `initialize` to set the current value using other functions; the initial value is not modified (it is saved and restored).

`virtual State* clone() const = 0;`

Derived state classes override this method to create an identical object to the one the method is called on.

`StringList print(int` *`verbose`*) const;

Output all info. This is NOT redefined for each type of state.

`bitWord attributes() const;`

Return my attribute bits.

`bitWord setAttributes(const Attribute&` *`attr`*);
`bitWord clearAttributes(const Attribute&` *`attr`*);

Set or clear attributes.

`const State* lookup(const char*` *`name`*`, Block*` *`b`*`);`

This method searches for a state named *`name`* in Block *`b`* or one of its ancestors, and either returns it or a null pointer if not found.

`int isA(const char*) const;`

This function returns true when given the name of the class or the name of any baseclass

9.1.2 The State parser and protected members

Most of the protected interface in the State class consists of a simple recursive-descent parser for parsing integer and floating expressions that appear in the initial value string. The ParseToken class represents tokens for this parser. It contains a token type (an integer code) and a token value, which is a union that represents either a character value, a string value, an integer value, a double value, a Complex value, or a State value (for use when the initializer references another state). Token types are equal to the ASCII character value for single-character tokens. Other possible token values are:

- `T_EOF` for end of file,
- `T_ERROR` for error,
- `T_Float` for a floating value,
- `T_Int` for an integer value,
- `T_ID` for a reference to a state, and
- `T_STRING` for a string value.

For most of these, the token value holds the appropriate value. Most derived State classes use this parser to provide uniformity of syntax and error reporting; however, it is not a requirement to use it. Derived `State` classes are expected to associate a `Tokenizer` object with their initial value string. The functions provided here can then be used to parse expressions appearing in that string.

`ParseToken getParseToken(Tokenizer&` *`tok`*`, int` *`stateType`* `= T_Float);`

This function obtains the next token from the input stream associated with the Tokenizer. If there is a pushback token, that token is returned instead. If it receives a '<' token, then it assumes that the next string delimited by white space is a file name. It substitutes references to other parameters in the filename and then uses the Tokenizer's include file capability to insert the contents of the file into the input stream. If it receives a '!' token, then it assumes that that the next string delimited by white space is a command to be evaluated by an external interpreter. It substitutes references to other parameters in the command, sends the resulting string to the interpreter defined by interp member described above for evaluation, and inserts the result into the input stream. The information both read from an external file and returned from an external interpreter is also parsed by this function. Therefore, the external interpreter can perform both numeric and symbolic computations. When the parser hits the end of the input stream, it returns T_EOF.

The characters in the set `[]+*-/()^` are considered to be special and the lexical value is equal to the character value. Integer and floating values are recognized and evaluated to produce either T_Int or T_Float tokens. However, the decision is based on the value of *`stateType;`* if it is T_Float, all numeric values are returned as T_Float; if it is T_Int, all numeric values are returned as T_Int. Names that take the form of a C or C++ identifier are assumed to be names of states defined at a higher level (states belonging to the parent galaxy or some ancestor galaxy). They are searched for using `lookup;` if not found, an error is reported using `parseError` and an error token is returned. If a State is found, a token of type T_ID is returned if it is an array state or COMPLEX; otherwise the state's current value is substituted and reparsed as a token. This means, for example, that a name of an IntState will be replaced with a T_Int token with the correct value.

`void parseError (const char*` *`part1`*`, const char*` *`part2`* `= "");`

This method produces an appropriately formatted error message with the name of the state and the arguments and calls `Error::abortRun.`

```
static ParseToken pushback();
static void setPushback(const ParseToken&);
static void clearPushback();
```

These functions manipulate the pushback token, for use in parsing. The first function returns the current pushback token, the second sets it to be a copy of the argument, the third clears it. There is only one such token, so the state parser is not reentrant.

`ParseToken evalIntExpression(Tokenizer&` *`lexer`*`);`
`ParseToken evalIntTerm(Tokenizer&` *`lexer`*`);`
`ParseToken evalIntFactor(Tokenizer&` *`lexer`*`);`
`ParseToken evalIntAtom(Tokenizer&` *`lexer`*`);`

These four functions implement a simple recursive-descent expression parser. An expression is either a term or a series of terms with intervening '+' or '-' signs. A term is either a factor or a series of factors with interventing '*' or '/' signs. A factor is either an atom or a series of atoms with intervening '^' signs for exponentiation. (Note, C fans! ^ means exponentiation, not exclusive-or!). An atom is any number of optional unary minus signs, followed either by a parenthesized expression or a T_Int token. If any of these methods reads too far, the pushback token is used. All `getParseToken` calls use *`stateType`* T_Int, so any floating values in the expression are truncated to integer. The token types returned from each of these methods will be one of T_Int, T_EOF, or T_ERROR.

`ParseToken evalFloatExpression(Tokenizer&` *`lexer`*`);`
`ParseToken evalFloatTerm(Tokenizer&` *`lexer`*`);`
`ParseToken evalFloatFactor(Tokenizer&` *`lexer`*`);`
`ParseToken evalFloatAtom(Tokenizer&` *`lexer`*`);`

These functions have the identical structure as the corresponding Int functions. The token types returned from each of these methods will be one of T_Float, T_EOF, or T_ERROR.

`InvokeInterp interp;`

An external interpreter for evaluating commands in a parameter definition preceded by the ! character and surrounded in quotes. By default, no interpreter is defined. If the interpreter were defined as the Tcl interpreter, then ! "expr abs(cos(1.0))" would compute 0.540302. Other parameters can be referenced as usual by using curly braces, e.g. ! "expr abs(cos({gain}))".

`StringList parseFileName(const char*);`

This method parses filenames that have been inherited from state values enclosed in curly braces.

`StringList parseNestedExpression(const char*` *`expression`*`);`

This method parses nested sub-expressions appearing in the *`expression`* , e.g. {{{FilterTapFile}/{File}}}, that might be passed off to another interpreter for evaluation, e.g. Tcl.

`Int mergeFileContents(Tokenizer&` *`lexer`*`, char*` *`token`*`);`

This method treats the next token on the *`lexer`* as a filename.

`Int sendToInterpreter(Tokenizer&` *`lexer`*`, char*` *`token`*`);`

This method sends the next token on the *`lexer`* to be evaluated by an external interpreter.

`Int getParameterName(Tokenizer& lexer, char* token);`

This method looks for parameters of the form `{name}`.

9.2 Types of states

9.2.1 Class IntState and class FloatState

Class `IntState`, derived from `State`, has an integer current value. Its `initialize()` function uses the `evalIntExpression` function to read an integer expression from the initial value string. If successful, it attempts to read another token from the string; if there is another token, it reports the error “extra text after valid expression”. An assignment operator is provided that accepts an integer value and loads it into the current value. A cast to integer is also defined for accessing the current value. The virtual function `currentValue` is overloaded to return a printed version of the current value. In addition to the `setInitValue` from class State, a second form is provided that takes an integer argument. Standard overrides for `isA`, `className`, and `clone` are provided. Class `FloatState` is almost identical to class IntState except that its data field is a double precision value; where IntState functions have an argument or return value of `int`, FloatState has a corresponding argument or return value of `double`. Both are generated from the same pseudo-template files. The `type()` function for IntState returns `"INT"`. For FloatState, `"FLOAT"` is returned. For both implementations, a prototype object is added to the KnownState list.

9.2.2 Class ComplexState

ComplexState is much like FloatState and IntState, except in the expressions it accepts for initial values. Its data member is Complex and it accordingly defines an assignment operator that takes a complex value and a conversion operator that returns one. The initial value string for a ComplexState takes one of three forms: it may be the name of a galaxy ComplexState, a floating expression (of the form accepted by `State::evalFloatExpression`), or a string of the form (*`floatexp1`* , *`floatexp2`*) where both *`floatexp1`* and *`floatexp2`* are floating expressions. For the second form, the imaginary part will always be zero. For the third form, the first expression gives the real part and the second gives the imaginary part.

9.2.3 Class StringState

A StringState’s current value is a string (more correctly, of type `const char*`). The current value is created by the `initialize()` function by scanning the initial value string. This string is copied literally, except that curly braces are special. If a pair of curly braces surrounds the name of a galaxy state, the printed representation of that state’s current value (returned by the `currentValue` function) is substituted. To get a literal curly brace in the current value, prefix it with a backslash. Class StringState defines assignment operators so that different string values can be copied to the current value; the value is copied with `saveString` and deleted by the destructor.

9.2.4 Numeric array states

Classes IntArrayState and FloatArrayState are produced from the same pseudo-template. Class ComplexArrayState has a very similar design. All return `TRUE` to `isArray`, provide an array element selection operator (`operator[](int)`), and an operator that converts the state into a pointer to the first element of its data (much like arrays in C). The expression parser for FloatArrayState accepts a series of “subarray expressions”, which are concatenated together to get the current value when `initialize()` is called. Subarray expressions may specify a single element, some number of copies of a single element, or a galaxy array state of the same type (another FloatArrayState). A single element specifier may either be a floating point value, a scalar (integer or floating) galaxy state name, or a general floating expression enclosed in parentheses. A number of copies of this single element can be specified by appending an integer expression enclosed in square brackets. The expression parsers for IntArrayState and ComplexArrayState differ only that where FloatArrayState wants a floating expression, IntArrayState wants an integer expression and ComplexArrayState wants a complex expression (an expression suitable for initializing a ComplexState).

9.2.5 Class StringArrayState

As its name suggests, the current value for a StringArrayState is an array of strings. White space in the initial value string separates “words”, and Each word is assigned by `initialize()` into a separate array element. Quotes can be use to permit “words” to have white space. Current values of galaxy states can be converted into single elements of the StringArrayState value by surrounding their names with curly braces in the initial value. Galaxy StringArrayState names will be translated into a series of values. There is currently no provision for modifying the current value of a StringArrayState other than calling of `initialize` to parse the current value string.

Chapter 10. Support for known lists and such

Authors: *Joseph T. Buck*

Other Contributors: *Neil Smyth*

Ptolemy is an extensible system, and in quite a few places it must create objects given only the name of that object. There are therefore several classes that are responsible for maintaining lists: the list of all known domains, of all known blocks, states, targets, etc. As a general rule, these classes support a `clone` or `makeNew` method to create a new object based on its name (you cannot clone a domain, however).

10.1 Class KnownBlock

The KnownBlock class is responsible for keeping a master list of all known types of Block objects in the system. All member functions of KnownBlock are static; the only non-static function of KnownBlock is the constructor. The KnownBlock constructor has the form

`KnownBlock(Block& `*`block`*`, const char* `*`name`*`);`

The only reason for constructing a KnownBlock object is for the side effects; the side effect is to add *`block`* to the known block list for its domain under the name *`name`*, using `addEntry`. The reason for using a constructor for this purpose is that constructors for global objects are called before execution of the main program; constructors therefore serve as a mechanism for execution of arbitrary initialization code for a module (as used here, “module” is an object file). Hence `ptlang`, the Ptolemy star preprocessor, generates code like the following for star definitions:

```
static XXXMyClass proto;
static KnownBlock entry(proto,"MyClass");
```

This code adds a prototype entry of the class to the known list. Dynamically constructed block types, such as interpreted galaxies, are added to the known list with a direct call to KnownBlock::addEntry. These cases should always supply an appropriate definition-source string so that conflicting block type definitions can be detected. KnownBlock keeps track of the source of the definition of every known block type. This allows compile.c to determine whether an Oct facet needs to be recompiled (without the source information, different facets that have the same base name could be mistaken for each other). This also allows us to generate some helpful warning messages when a block name is accidentally re-used. The source location information is currently rather crude for everything except Oct facets, but that’s good enough to generate a useful warning in nearly all cases. KnownBlock assigns a sequential serial number to each definition or redefinition of a known block type. This can be used, for example, to determine whether a galaxy has been compiled more recently than any of its constituent blocks.

`static void addEntry (Block & `*`block`*`, const char* `*`name`*`, int `*`onHeap,`*` const char* `*`definitionSource`*`);`

This function actually adds the block to the list. Separate lists are maintained for each domain; the block is added to the list corresponding to '`block.domain()`'. If *`onHeap`* is true, the block will be destroyed when the entry is removed or replaced from the list. definitionSource should be `NULL` for any block type defined by C++ code (this is what is passed by the KnownBlock constructor). It should be a hashstring'ed path name for a block defined by an identifiable file (such as an Oct facet), or a special case constant string for other cases such as the ptcl `defgalaxy` command.

`static const Block* find (const char* `*`name`*, const char* *`dom`*);

The find method returns a pointer the appropriate block in the specified domain. A null pointer is returned if no match is found.

`static Block* clone (const char* `*`name`*, const char* *`dom`*);
`static Block* makeNew (const char* `*`name`*, const char* *`dom`*);

The `clone` method takes a string, finds the appropriate block in the specified domain, and returns a clone of that block (the `clone` method is called on the block. This method, as a rule, generates a duplicate of the block. The `makeNew` function is similar except that `makeNew` is called on the found block. As a rule, `makeNew` returns an object of the same class, but with default initializations (for example, with default state values). For either of these, an error message is generated (with `Error::abortRun`) and a null pointer is returned if there is no match. To avoid a crash in the event of a self-referential galaxy definition, recursive clone or makeNew attempts are detected, and are terminated by generating an error message and returning a null pointer.

`static StringList nameList (const char* `*`domain`*);

Return the names of known blocks in the given domain (second form). Names are separated by newline characters.

`static const char* defaultDomain ();`

Returns the default domain name. This is not used internally for anything; it is just set to the first domain seen during the building of known lists.

`static int setDefaultDomain (const char* `*`newval`*`);`

Set the default domain name. Return `FALSE` if the specified value is not a valid domain.

`static int validDomain (const char* `*`newval`*`);`

Return `TRUE` if the given name is a valid domain.

`static int isDynamic (const char* `*`type`*`, const char* `*`dom`*`);`

Return TRUE if the named block is dynamically linked. There is an iterator associated with KnownBlock, called KnownBlockIter. It takes as an argument the name of a domain. The argument may be omitted, in which case the default domain is used. Its `next` function returns the type `const Block*`; it steps through the blocks on the known list for that domain.

`static int isDefined (const char*` *`type`*`, const char*` *`dom`*`,`
`const char*` *`definitionSource`*`);`

If there is a known block of the given name and domain, return TRUE and pass back its definition source string.

`static long serialNumber (const char*` *`name`*`, const char*` *`dom`*`);`

Look up a KnownBlock definition by name and domain, and return its serial number. Returns 0 iff no matching definition exists.

`static long serialNumber (Block&` *`block`*`);`

Given a block, find the matching KnownBlock definition, and return its serial number (or 0 if no matching definition exists).

10.2 Class KnownTarget

The KnownTarget class keeps track of targets in much the same way that KnownBlock keeps track of blocks. There are some differences: there is only a single list of targets, not one per domain as for blocks. The constructor works exactly the same way that the constructor for KnownBlock works; the code

```
    static MyTarget proto(args);
    static KnownTarget entry(proto,"MyTarget");
```

adds the prototype instance to the known list with a call to `addEntry.`

static void addEntry (Target *`&target`*, const char* *`name`*, int *`onHeap`*);

This function actually adds the Target to the list. If *`onHeap`* is true, the target will be destroyed when the entry is removed or replaced from the list. There is only one list of Targets.

`static const Target* find (const char*` *`name`*`);`

The find method returns a pointer the appropriate target. A null pointer is returned if no match is found.

`static Target* clone (const char*` *`name`*`);`

The `clone` method takes a string, finds the appropriate target on the known target list, and returns a clone of that target (the `cloneTarget` method is called on the target). This method, as a rule, generates a duplicate of the target. An error message is generated (with `Error::abortRun)` and a null pointer is returned if there is no match.

`static int getList (const Block&` *`b`*`, const char**` *`names`*`, int` *`nMax`*`);`

This function returns a list of names of targets that are compatible with the Block *`b.`* The return value gives the number of matches. The *`names`* array can hold *`nMax`* strings; if there are more, only the first *`nMax`* are returned.

`static int getList (const char*` *`dom`*`, const char**` *`names`*`, int` *`nMax`*`);`

This function is the same as above, except that it returns names of targets that are compatible with stars of a particular domain.

`static int isDynamic (const char* *type*);`

Return true if there is a target on the known list named *type* that is dynamically linked; otherwise return false.

`static const char* defaultName (const char* *dom* = 0);`

Return the default target name for a domain (default: current domain). There is an iterator associated with KnownTarget, called KnownTargetIter. Since there is only one known target list, it is unusual for an iterator in that it takes no argument for its constructor. Its `next` function returns the type `const Target *`; it steps through the targets on the known list.

10.3 Class Domain

The Domain class represents the information that Ptolemy needs to know about a particular domain so that it can create galaxies, wormholes, nodes, event horizons, and such for that domain. For each domain, the designer creates a derived class of Domain and one prototype object. Thus the Domain class has two main parts: a static interface, which manages access to the list of Domain objects, and a set of virtual functions, which provides the standard interface for each domain to describe its requirements.

10.3.1 Domain virtual functions

`virtual Star& newWorm (Galaxy& *innerGal*, Target* *innerTarget* = 0);`

This function creates a new wormhole with the given inner galaxy and inner target. The default implementation returns an error. XXXDomain might override this as follows:

```
Star& XXXDomain::newWorm(Galaxy& innerGal,Target* innerTar-
get)  {
      LOG\_NEW; return *new XXXWormhole(innerGal,innerTarget);
}
```

```
virtual EventHorizon& newFrom();
virtual EventHorizon& newTo();
```

These functions create event horizon objects to represent the XXXfromUniversal and XXXtoUniversal functions. The default implementations return an error. XXXDomain might override these as

```
EventHorizon& XXXDomain::newFrom() {
    LOG_NEW; return *new XXXfromUniversal;
}
EventHorizon& XXXDomain::newTo() {
    LOG_NEW; return *new XXXtoUniversal;
}
```

`virtual Geodesic& newGeo(int *multi*=FALSE);`

This function creates a new geodesic for point-to-point connection or a “node” suitable for multi-point connections.

`virtual int isGalWorm();`

This function returns `FALSE` by default. If overridden by a function that returns `TRUE`, a wormhole will be created around every galaxy for this domain.

`virtual const char* requiredTarget();`

If non-null, this method returns requirement for targets for use with this domain.

10.4 Class KnownState

KnownState manages two lists of states, one to represent the types of states known to the system (integer, string, complex, array of floating, etc.), and one to represent certain predeclared global states. It is very much like KnownBlock in internal structure. Since it manages two lists, there are two kinds of constructors.

`KnownState (State` *`&state`*`, const char*` *`name`*`);`

This constructor adds an entry to the state type list. For example,

```
static IntState proto;
static KnownState entry(proto,"INT");
```

permits IntStates to be produced by cloning. The *`type`* argument must be in upper case, because of the way `find` works (see below). The second type of constructor takes three arguments:

`KnownState (State` *`&state`*`, const char*` *`name`*`, const char*` *`value`*`);`

This constructor permits names to be added to the global state symbol list, for use in state expressions. For example, we have

```
static FloatState pi;
KnownState k_pi(pi,"PI","3.14159265358979323846");
```

`static const State* find (const char*` *`type`*`);`

The `find` method returns a pointer the appropriate prototype state in the state type list. The argument is always changed to upper case. A null pointer is returned if there is no match.

`static const State* lookup (const char*` *`name`*`);`

The lookup method returns a pointer to the appropriate state in the global state list, or null if there is no match.

`static State* clone (const char*` *`type`*`);`

The clone method takes a string, finds the appropriate state using `find,` and returns a clone of that block. A null pointer is returned if there is no match, and `Error::error` is also called.

`static StringList nameList();`

Return the names of all the known state types, separated by newlines.

`static int nKnown();`

Return the number of known states.

Chapter 11. I/O classes

Authors: *Joseph T. Buck*

Other Contributors: *Bilung Lee*

11.1 StringList, a kind of String class

Class `StringList` provides a mechanism for organizing a list of strings. It can also be used to construct strings of unbounded size, but the class `InfString` is preferred for this. It is privately derived from `SequentialList`. Its internal implementation is as a list of `char*` strings, each on the heap. A `StringList` object can be treated either as a single string or as a list of strings; the individual substrings retain their separate identity until the conversion operator to type `const char*` is invoked. There are also operators that add numeric values to the `StringList`; there is only one format available for such additions. WARNING: if a function or expression returns a `StringList`, and that value is not assigned to a `StringList` variable or reference, and the `(const char*)` cast is used, it is possible (likely under g++) that the `StringList` temporary will be destroyed too soon, leaving the `const char*` pointer pointing to garbage. Always assign a temporary `StringList` to a `StringList` variable or reference before using the `const` char* conversion. Thus, instead of

```
function_name(xxx,(const char*)functionReturningStringList(),yyy);
```

one should use

```
StringList temp_name = (const char*)functionReturningStringList();
function_name(xxx,temp_name,yyy);
```

This includes code like

```
strcpy(destBuf,functionReturningStringList());
```

which uses the `const char*` conversion implicitly.

11.1.1 StringList constructors and assignment operators

The default constructor makes an empty `StringList`. There is also a copy constructor and five single-argument constructors that can function as conversions from other types to type `StringList`; they take arguments of the types `char`, `const char *`, `int`, `double`, and `unsigned int`. There are also six assignment operators corresponding to these constructors: one that takes a `const StringList&` argument and also one for each of the five standard types: `char`, `const char *`, `int`, `double`, and `unsigned int`. The resulting object has one piece, unless initialized from another `StringList` in which case it has the same number of pieces.

11.1.2 Adding to StringLists

There are six functions that can add a printed representation of an argument to a `StringList`: one each for arguments of type `const StringList&`, `char`, `const char *`, `int`, `dou-`

`ble`, and `unsigned int`. In each case, the function can be accessed in either of two equivalent ways:

```
StringList& operator += (type arg);
StringList& operator << (type arg);
```

The second "stream form" is considered preferable; the "+=" form is there for backward compatibility. If a `StringList` object is added, each piece of the added `StringList` is added separately (boundaries between pieces are preserved); for the other five forms, a single piece is added.

11.1.3 StringList information functions

```
const char* head() const;
```

Return the first substring on the list (the first "piece"). A null pointer is returned if there are none.

```
int length() const;
```

Return the length in characters.

```
int numPieces() const;
```

Return the number of substrings in the `StringList`.

11.1.4 StringList conversion to const char *

```
operator const char* ();
```

This function joins all the substrings in the `StringList` into a single piece, so that afterwards `numPieces` will return 1. A null pointer is always returned if there are no characters. Warning: if this function is called on a temporary `StringList`, it is possible that the compiler will delete the `StringList` object before the last use of the returned `const char *` pointer. The result is that the pointer may wind up pointing to garbage. The best way to work around such problems is to make sure that any `StringList` object "has a name" before this conversion is applied to it; e.g., assign the results of functions returning `StringList` objects to local `StringList` variables or references before trying to convert them.

```
char* newCopy() const;
```

This function makes a copy of the `StringList`'s text in a single piece as a `char*` in dynamic memory. The object itself is not modified. The caller is responsible for deletion of the returned text.

11.1.5 StringList destruction and zeroing

```
void initialize();
```

This function deallocates all pieces of the `StringList` and changes it to an empty `StringList`.

```
~StringList();
```

The destructor calls the `initialize` function.

11.1.6 Class StringListIter

Class `StringListIter` is a standard iterator that operates on `StringLists`. Its `next()` function returns a pointer of type `const char*` to the next substring of the `StringList`. It is important to know that the operation of converting a `StringList` to a `const char*` string joins all the substrings into a single string, so that operation should be avoided if extensive use of `StringListIter` is planned.

11.2 InfString, a class supporting unbounded strings

Class `InfString` provides a mechanism for building strings of unbounded size. It provides a subset of the functions in a typical C++ String class. Strings can be built up piece by piece. As segments are added, they are copied, so the caller need not keep the segments around. Upon casting to `(char*),` the strings are collapsed into one continuous string, and a pointer to that string is returned. The calling function can treat this as an ordinary pointer to an ordinary array of characters, and can modify the characters. But the length of the string should not be changed, nor should the string be deleted. The `InfString` destructor is responsible for freeing the allocated memory. `InfString` is publically derived from `StringList`, adding only the cast `char*`. Its internal implementation is as a list of `char*` strings, each on the heap. The individual substrings retain their separate identity until the conversion cast to type `char*` is invoked, although if access to the individual strings is needed, then `StringList` should be used. There are also operators that add numeric values to the `StringList`; there is only one format available for each such addition. WARNING: if a function or expression returns an `InfString`, and that value is not assigned to an `InfString` variable or reference, and the `(char*)` cast is used, it is possible (likely under g++) that the `InfString` temporary will be destroyed too soon, leaving the `char*` pointer pointing to garbage. Always assign temporary `InfString` to `InfString` variables or references before using the `char*` conversion. Thus, instead of

```
function_name(xxx,(char*)functionReturningInfString(),yyy);
```

one should use

```
InfString temp_name = (char*)functionReturningInfString();
function_name(xxx,temp_name,yyy);
```

This includes code like

```
strcpy(destBuf,functionReturningInfString());
```

which uses the `char*` conversion implicitly.

11.2.1 InfString constructors and assignment operators

The default constructor makes an empty `InfString`. There is also a copy constructor and six single-argument constructors that can function as conversions from other types to type `InfString`; they take arguments of the types `char`, `const char*`, `int`, `double`, `unsigned int`, and `const StringList&`. There are also seven assignment operators corresponding to these constructors: one that takes a `const InfString&` argument and also one for each of the six standard types: `char`, `const char*`, `int`, `double`, `unsigned int`, and `const StringList&`.

11.2.2 Adding to InfStrings

There are seven functions that can add a printed representation of an argument to a InfString: one each for arguments of type `const InfString&`, `char`, `const char*`, `int`, `double`, `unsigned int`, and `const StringList&`. In each case, the function can be accessed in either of two equivalent ways:

```
InfString& operator += (type arg);
InfString& operator << (type arg);
```

The second “stream form” is considered preferable; the “+=” form is there for backward compatibility. If a `InfString` object is added, each piece of the added `InfString` is added separately (boundaries between pieces are preserved); for the other five forms, a single piece is added.

11.2.3 InfString information functions

```
int length() const;
```

Return the length in characters.

11.2.4 InfString conversion to char *

```
operator char* ();
```

This function joins all the substrings in the `InfString` into a single piece, a returns a pointer to the resulting string. A null pointer is always returned if there are no characters. Warning: as pointed out above, if this function is called on a temporary `InfString`, it is possible that the compiler will delete the `InfString` object before the last use of the returned `char*` pointer. The result is that the pointer may wind up pointing to garbage. The best work-around for such problems is to make sure that any `InfString` object “has a name” before this conversion is applied to it; e.g. assign the results of functions returning `InfString` objects to local `InfString` variables or references before trying to convert them.

```
char* newCopy() const;
```

This function makes a copy of the `InfString`’s text in a single piece as a `char*` in dynamic memory. The `InfString` object itself is not modified. This is useful when the caller wishes to be responsible for deletion of the returned text.

11.2.5 InfString destruction and zeroing

```
void initialize();
```

This function deallocates all pieces of the `InfString` and changes it to an empty `InfString`.

```
~InfString();
```

The destructor calls the `initialize` function.

11.2.6 Class InfStringIter

Class `InfStringIter` is a standard iterator that operates on `InfStrings`. However, the `InfString` class is not intended for use when access to the individual components of the string is desired. Use `StringList` for this.

11.3 Tokenizer, a simple lexical analyzer class

The `Tokenizer` class is designed to accept input for a string or file and break it up into tokens. It is similar to the standard istream class in this regard, but it has some additional facilities. It permits character classes to be defined to specify that certain characters are white space and others are "special" and should be returned as single-character tokens; it permits quoted strings to override this, and it has a file inclusion facility. In short, it is a simple, reconfigurable lexical analyzer. `Tokenizer` has a public const data member named `defWhite` that contains the default white space characters: space, newline, and tab. It is possible to change the definition of white space for a particular constructor.

11.3.1 Initializing Tokenizer objects

`Tokenizer` provides three different constructors:

`Tokenizer();`

The default constructor creates a `Tokenizer` that reads from the standard input stream, `cin.` Its special characters are simply \key (and \key).

```
Tokenizer(istream& input,const char* spec,
          const char* w = defWhite);
```

This constructor creates a `Tokenizer` that reads from the stream named by *`input.`* The other arguments specify the special characters and the white space characters.

```
Tokenizer(const char* buffer,const char* spec,
          const char* w = defWhite);
```

This constructor creates a `Tokenizer` that reads from the null-terminated string in *`buffer.`* `Tokenizer`'s destructor closes any include files associated with the constructor and deletes associated internal storage. The following operations change the definition of white space and of special characters, respectively:

```
const char* setWhite(const char* w);
const char* setSpecial(const char* s);
```

In each case, the old value is returned. By default, the line comment character for `Tokenizer` is #. It can be changed by

`char setCommentChar(char n);`

Use an argument of 0 to disable the feature. The old comment character is returned.

11.3.2 Reading from Tokenizers

The next operation is the basic mechanism for reading tokens from the `Tokenizer`:

`Tokenizer& operator >> (char* pBuffer);`

Here *`pBuffer`* points to a character buffer that reads the token. There is a design flaw: there isn't a way to give a maximum buffer length, so overflow is a risk. By analogy with streams, the following operation is provided:

`operator void*();`

It returns null if `EOF` has already been reached and non-null otherwise. This permits loops like

```
Tokenizer tin;
while (tin) { ... do stuff ... }
```

`int eof() const;`

Returns true if the end of file or end of input has been reached on the `Tokenizer`. It is possible that there is nothing left in the input but write space, so in many situations `skipwhite` should be called before making this test.

`void skipwhite();`

Skip white space in the input.

`void flush();`

If in an include file, the file is closed. If at the top level, discard the rest of the current line.

11.3.3 Tokenizer include files

`Tokenizer` can use include files, and can nest them to any depth. It maintains a stack of include files, and as `EOF` is reached in each file, it is closed and popped off of the stack. The method

`int fromFile(const char*` *name*);

opens a new file and the `Tokenizer` will then read from that. When that file ends, `Tokenizer` will continue reading from the current point in the current file.

`const char* current_file() const;`
`int current_line() const;`

These methods report on the file name and line number where `Tokenizer` is currently reading from. This information is maintained for include files. At the top level, `current_file` returns a null pointer, but `current_line` returns one more than the number of line feeds seen so far.

`int readingFromFile() const;`

Returns true (1) if the `Tokenizer` is reading from an include file, false (0) if not.

11.4 pt_ifstream and pt_ofstream: augmented fstream classes

The classes `pt_ifstream` and `pt_ofstream` are derived from the standard stream classes `ifstream` and `ofstream`, respectively. They are defined in the header file `pt_fstream.h.` They add the following features: First, certain special “filenames” are recognized. If the filename used in the constructor or an `open` call is `cin>`, `cout>`, `cerr>`, or `clog>` (the angle brackets must be part of the string), then the corresponding standard stream of the same name is used for input (`pt_ifstream`) or output (`pt_ofstream`). In addition, C standard I/O fans can specify `stdin>`, `stdout>`, or `stderr>` as well. Second, the Ptolemy `expandPathName` is applied to the filename before it is opened, permitting it to start with `~user` or `$VAR.` Finally, if a failure occurs when the file is opened, `Error::abortRun` is called with an appropriate error message, including the Unix error condition. Otherwise these classes are identical to the standard ifstream and ofstream classes and can be used as replacements.

11.5 XGraph, an interface to the xgraph program

The `XGraph` class provides an interface for the `xgraph` program for plotting data on an X window system display. The modified `xgraph` program provided with the Ptolemy distribution should be used, not the contributed version from the X11R5 tape. The constructor for `XGraph` does not completely initialize the object; initialization is completed by the `initialize()` method:

```
void initialize(Block* parent, int noGraphs,
     const char* options, const char* title,
     const char* saveFile = 0, int ignore = 0);
```

The *`parent`* argument is the name of a `Block` that is associated with the `XGraph` object; this `Block` is used in `Error::abortRun` messages to report errors. *`noGraphs`* specifies the number of data sets that the graph will contain. Each data set is a separate stream and is plotted in a different color (a different line style for B/W displays). *`options`* is a series of command line options that will be passed unmodified to the `xgraph` program. It is subject to expansion by the Unix shell. *`title`* is the title for the graph; it can contain special characters (it is *not* subjected to expansion by the Unix shell). *`saveFile`* is the name of a file to save the graph data into, in ASCII form. If it is not given, the data are not saved, and a faster binary format is used. *`ignore`* specifies the number of initial points to ignore from each data set.

```
void setIgnore(int n);
```

Reset the “ignore” parameter to *`n.`*

```
void addPoint(float y);
```

Add a single point to the first data set whose X value is automatically generated (0, 1, 2, 3... on successive calls) and whose Y value is *`y.`*

```
void addPoint(float x,float y);
```

Add the point (*`x, y)`* to the first data set.

```
void addPoint(int dataSet,float x,float y);
```

Add the point (*`x, y)`* to the data set indicated by *`dataSet.`* Data sets start with 1.

```
void newTrace(int dataSet = 1);
```

Start a new trace for the nth dataset. This means that there will be no connecting line between the last point plotted and the next point plotted.

```
void terminate();
```

This function flushes the data out to disk, closes the files, and invokes the `xgraph` program. If the destructor is called before `terminate,` it will close and delete the temporary files.

11.6 Histogram classes

The `Histogram` class accepts a stream of data and accumulates a histogram. The `XHistogram` class uses a `Histogram` to collect the data and an `XGraph` to display it.

11.6.1 Class Histogram

The `Histogram` class accumulates data in a histogram. Its constructor is as follows:

```
Histogram(double width = 1.0, int maxBins = HISTO_MAX_BINS);
```

The default maximum number of bins is 1000. The bin centers will be at integer multiples of the specified bin width. The total width of the histogram depends on the data; however, there will always be a bin that includes the first point.

```
void add(double x);
```

Add the point *x* to the histogram.

```
int numCounts() const;
double mean() const;
double variance() const;
```

Return the number of counts, the mean, and the variance of the data in the histogram.

```
int getData(int binno, int& count, double& binCenter);
```

Get counts and bin centers by bin number, where 0 indicates the smallest bin. Return `TRUE` if this is a valid bin. Thus the entire histogram data can be retrieved by stepping from 0 to the first failure.

11.6.2 Class XHistogram

An `XHistogram` object has a private `XGraph` member and a private `Histogram` member. The functions

```
int numCounts() const;
double mean() const;
double variance();
```

simply pass through to the `Histogram` object, and

```
void addPoint(float y);
```

adds a point to the histogram and does other bookkeeping. There are two remaining methods:

```
void initialize(Block* parent, double binWidth,
   const char* options, const char* title,
   const char* saveFile, int maxBins = HISTO_MAX_BINS
```

This method initializes the graph and histogram object. *`parent`* is the parent `Block`, used for error messages. *`binWidth`* and *`maxBins`* initialize the `Histogram` object. *`options`* is a string that is included in the command line to the `xgraph` program; other options, including `-bar -nl -brw` value*,* are passed as well. *`title`* is the graph title, and *`saveFile`,* if non-null, gives a file in which the histogram data is saved (this data is the histogram counts, not the data that was input with `addPoint).`

```
void terminate();
```

This method completes the histogram, flushes out the temporary files, and executes `xgraph`.

Chapter 12. Miscellaneous classes

Authors: *Joseph T. Buck*

Other Contributors: *Yuhong Xiong*

This section includes classes that did not fit elsewhere.

12.1 Mathematical classes

12.1.1 Class Complex

Class Complex is a simple subset of functions provided in the Gnu and AT&T complex classes. The standard arithmetic operators are implemented, as are the assignment arithmetic operators `+=`, `-=`, `*=`, and `/=`, and equality and inequality operators `==` and `!=`. There is also `real()` and `imag()` methods for accessing real and imaginary parts. It was originally written when libg++ was subject to the GPL. The current licensing for libg++ does not prevent us from using it and still distributing Ptolemy the way we want, but having it makes ports to other compilers (e.g. cfront) easier. The following non-member functions take Complex arguments:

```
Complex conj(const Complex& arg);
double real(const Complex& arg);
double imag(const Complex& arg);
double abs(const Complex& arg);
```
Return the conjugate, real part, imaginary part, or absolute value, respectively.

```
double arg(const Complex& arg);
```
Return the angle between the X axis and the vector made by the argument. The expression `abs(z)*exp(Complex(0.,1.)*arg(z))` is in theory always equal to z.

```
double norm(const Complex& arg);
```
return the absolute value squared.

```
Complex sin(const Complex& arg);
Complex cos(const Complex& arg);
Complex exp(const Complex& arg);
Complex log(const Complex& arg);
Complex sqrt(const Complex& arg);
```
Standard mathematical functions. `log` returns the principal logarithm.

```
Complex pow(double base,const Complex& expon);
xpon);
```
Raise base to expon power. There is also an operator to print a Complex on an ostream.

12.1.2 class Fraction

Class Fraction represents fractions. The header `Fraction.h` also provides declarations for the `lcm` (least common multiple) and `gcd` (greatest common divisor) functions, as these functions are needed for Fraction but are generally useful.

```
Fraction ();
Fraction (int num, int den=1);
```

The default constructor produces a fraction with numerator 0 and denominator 1. The other constructor allows the numerator and denominator to be specified arbitrarily.

```
int num() const;
int den() const;
```

Return the numerator or denominator.

```
operator double() const;
```

Return the value of the fraction as a double. Class Fraction implements the basic binary math operators `+, -, *, /;` the assignment operators `=, +=, -=, *=,` and `/=,` and the equality test operators == and `!=.` The method

```
Fraction& simplify();
```

reduces the fraction to lowest terms, and returns a reference to the fraction. There is also an operator to print a Fraction on an ostream.

12.2 Class IntervalList

The IntervalList class represents a set of unsigned integers, represented as a series of intervals of integers that belong to the set. It is built on top of a class Interval that represents a single interval. There is also a text representation for IntervalLists. This representation can be used to read or write IntervalList objects to streams, and also can be used in the IntervalList constructor. This text representation looks exactly like the format the “rn” newsreader uses to record which articles have been read in a Usenet newsgroup (which is where we got it from; thank you, Larry Wall). In the text representation, an IntervalList is specified as one or more Intervals, separated by commas. An Interval is either an unsigned integer or two unsigned intervals with an intervening minus sign. Here is one possible IntervalList specification: 1-1003,1006,1008-1030,1050 White space is not permitted in this representation. IntervalList specifiers do not have to be in increasing order, but if they are not, they are changed to "canonical form", in which any overlapping intervals are merged and the intervals are sorted to appear in increasing order. An IntervalList is best thought of as a set of unsigned integers. Its methods in many cases perform set operations: forming the union, intersection, or set difference of two IntervalLists.

12.2.1 class Interval and methods

The Interval class is in some ways simply an implementation detail of class IntervalList, but since its existence is exposed by public methods, it is documented here. An Interval has an *`origin`* and a *`length,`* and represents the set of *`length`* unsigned integers beginning with *`origin.`* It also has a pointer that can point to another Interval. The constructor

```
Interval(unsigned origin=0, unsigned length=0,
    Interval* nxt = 0);
```

permits all these members to be set. The copy constructor copies the origin and length values but always sets the next pointer to null. A third constructor

```
Interval(const Interval& i1,Interval* nxt);
```

permits a combination of a copy and a next-pointer initialization. The members

```
unsigned origin() const;
unsigned length() const;
```

return the origin and length values.

```
unsigned end() const;
```

The `end` function returns the last unsigned integer that is a member of the Interval; 0 is returned for empty Intervals. There are a number of queries that are valuable for building a set class out of Intervals:

```
int isAfter(const Interval &i1) const;
```

`isAfter` returns true if this Interval begins after the end of interval *i1* .

```
int endsBefore(const Interval &i1) const;
```

`endsBefore` returns true if this Interval ends strictly before the origin of interval i1.

```
int completelyBefore(const Interval &i1) const;
```

`completelyBefore` returns true if `endsBefore` is true and there is space between the intervals (they cannot be merged).

```
int mergeableWith(const Interval& i1) const;
```

`mergeableWith` returns true if two intervals overlap or are adjacent, so that their union is also an interval.

```
int intersects(const Interval& i1) const;
```

`intersects` returns true if two intervals have a non-null intersection.

```
int subsetOf(const Interval& i1) const;
```

`subsetOf` returns true if the argument is a subset of this interval.

```
void merge(const Interval& i1);
```

`merge` alters the interval to the result of merging it with *i1.* It is assumed that `mergeableWith` is true.

```
Interval& operator&=(const Interval& i1);
```

This Interval is changed to the intersection of itself and of *i1.*

12.2.2 IntervalList public members

```
IntervalList();
```

The default constructor produces the empty IntervalList.

```
IntervalList(unsigned origin,unsigned length);
```

This constructor creates an IntervalList containing *`length`* integers starting with *`ori-gin.`*

`IntervalList(const char*` *`definition`*`);`

This constructor takes a definition of the IntervalList from the string in *`definition,`* parses it, and creates the list of intervals accordingly. There is also a copy constructor, an assignment operator, and a destructor.

`int contains(const Interval&` *`i1`*`) const;`

The `contains` method returns 0 if no part of *`i1`* is in the IntervalList, 1 if *`i1`* is completely contained in the IntervalList, and -1 if *`i1`* is partially contained (has a non-null intersection).

`IntervalList& operator|=(const Interval&` *`src`*`);`

Add a new interval to the interval list.

`IntervalList& operator|=(const IntervalList&` *`src`*`);`

Sets the IntervalList to the union of itself and *`src.`*

`IntervalList operator&(const IntervalList&` *`arg`*`) const;`

The binary `&` operator returns the intersection of its arguments, which are not changed.

`IntervalList& subtract(const Interval&` *`i1`*`);`
`IntervalList& operator-=(const Interval&` *`i1`*`);`

Subtract the Interval *`i1`* from the list. That is, any intersection is removed from the set. Both the `subtract` and `-=` forms are equivalent.

`IntervalList& operator-=(const IntervalList &`*`arg`*`);`

This one subtracts the argument *`arg`* from the list (removes any intersection).

`int empty() const;`

Return `TRUE` (1) for an empty IntervalList, otherwise `FALSE` (0).

12.2.3 IntervalList iterator classes.

There are two iterator classes associated with IntervalList, IntervalListIter and CIntervalListIter. The only difference is that the latter iterator can be used with const IntervalList objects and returns pointers to const Interval objects; the former requires a non-const IntervalIList and returns pointers to Interval. These objects obey the standard iterator interface; the `next()` or `++` function returns a pointer to the next contained Interval; `reset` goes back to the beginning.

12.3 Classes for interacting with the system clock

These classes provide simple means of interacting with the operating system's clock – sleeping until a specified time, timing events, etc. They may be replaced with something more general. Class TimeVal represents a time interval to microsecond precision. There are two constructors:

`TimeVal();`
`TimeVal(double` *`seconds`*`);`

The first represents a time interval of zero. In the second case, the `seconds` argument is rounded to the nearest microsecond. These classes rely on features found in BSD-based Unix systems and newer System V Unix systems. Older System V systems tend not to provide the ability to sleep for a time specified more accurately than a second.

```
operator double() const;
```

This returns the interval value as a double.

```
TimeVal operator +(const TimeVal& arg) const;
TimeVal operator -(const TimeVal& arg) const;
TimeVal& operator +=(const TimeVal& arg);
TimeVal& operator -=(const TimeVal& arg);
```

These operators do simple addition and subtraction of TimeVals.

```
int operator >(const TimeVal& arg) const;
int operator <(const TimeVal& arg) const;
```

These operators do simple comparisons of TimeVals.

Class Clock provides a simple interface to the system clock for measurement of actual elapsed time. It has an internal TimeVal field that represents the starting time of a time interval.

```
Clock();
```

The constructor creates a Clock with starting time equal to the time at which the constructor is executed.

```
void reset();
```

This method resets the start time to “now”.

```
TimeVal elapsedTime() const;
```

This method returns the elapsed time since the last `reset` or the call to the constructor.

```
int sleepUntil(const TimeVal& howLong) const;
```

This method causes the process to sleep until *`howLong`* after the start time.

Chapter 13. Overview of Parallel Code Generation

Authors: *Soonhoi Ha*

This chapter describes the overall procedure of parallel code generation in Ptolemy. We start with an SDF program graph and a multiprocessor target description. In the target definition, we specify the number of processors and some information about the processors with target parameters. If the number of processor is given 1, it is classified as a sequential code generation problem: a chosen SDF scheduler schedules the graph and code is generated based on the scheduling result. Parallel code generation is a bit more complicated.

If the number of processor is greater than 1, we create an APEG (acyclic precedence expanded graph) associated with the SDF program graph. The APEG graph displays all precedence relations between invocations of the SDF stars. All parallel schedulers take this APEG graph as an input graph and generate the schedule. In Ptolemy, we can have many scheduling algorithms (currently 3), and choose one by setting the appropriate target parameters. There is a common framework all parallel scheduling algorithm should be fit into . The scheduling result indicates the assignment and the ordering of star invocations in the processors. The next step is to generate code for each processor based on the scheduling result.

We create an SDF *sub-universe* for each processor. The sub-universe consists of stars assigned to the processor and some other automatically inserted stars, for example send and receive stars for interprocessor communication. We apply the sequential code generation routine for each processor with the associated sub-universe.

We may generate parallel code inside a wormhole so that the main workstation can communicate with the target multiprocessor system. Then, the wormhole interface code should be added to the generated code.

The following chapter will explain each steps in significant detail with code segments.

Chapter 14. APEG generation

Authors: *Soonhoi Ha*

Since all code generation domains depends on the SDF domain, and the same routine is needed by a specialized loop scheduler in the SDF domain ($PTOLEMY/src/domains/sdf/loopScheduler), the source of APEG generation is placed in $PTOLEMY/src/domains/sdf/kernel.

An APEG graph (an ExpandedGraph class) consists of EGNodes and EGGates. Class EGNode represents an object corresponding to an invocation of a DataFlowStar (DataFlowStar is a base class of SDFStar class). An EGNode has a list of EGGates. EGGate class is similar to PortHole class in the respect that it is an object for connection between EGNodes. Between two EGGates, there exists an EGArc object. All connections in an APEG graph is homogeneous. If there is a sample rate change on an arc in the SDF program graph, the arc is mapped to several homogeneous arcs. APEG generation routines are defined as member methods of the ExpandedGraph class.

Refer to “Class ExpandedGraph” on page 14-5, for he main discussion of APEG generation.

14.1 Class EGArc

Class EGArc contains the information of (1) sample rate of the arc and (2) the initial delay on the arc.

`EGArc(int` *`arc_samples`*, int *`arc_delay`*);

The constructor requires two arguments for sample rate and the number of initial delays on the arc.

```
int samples();
int delay();
```

These functions return the sample rate of the arc, and the initial delay on the arc. We can increase the sample rate of the arc using the following method

`void addSamples(int` *`increments`*);

There is no protected members in Class EGArc.

14.2 Class EGGate

Class EGGate is a terminal in an EGNode for connection with other EGNodes. A list of EGGates will become a member of EGNode, called `ancestors` or `descendants` based on the direction of connection.

14.2.1 EGGate public members

`EGGate(EGNode*` *`parent`*, PortHole* *`pPort`*);

Is a constructor. The first argument is the EGNode that this EGGate belongs to, and the second argument is the corresponding porthole of the original SDF graph.

`const PortHole* aliasedPort();`
`const char* name() const;`

The above methods returns the corresponding porthole of the original SDF graph, the name of the porthole.

`int isItInput();`

Returns TRUE or FALSE based on whether the corresponding porthole is an input or not.

`void allocateArc(EGGate*` *`dest`*, int *`no_samples`*, int *`no_delay`*);

The method creates a connection between this EGGate and the first argument by allocating an arc with information from the second and the third arguments. It should be called once per connection.

`int samples();`
`int delay();`
`void addSamples(int` *`increments`*);

These methods call the corresponding methods of the EGArc class if an arc was already allocated by `allocateArc.`

`EGGate* farGate();`
`EGNode* farEndNode();`
`DataFlowStar* farEndMaster();`
`int farEndInvocation();`

The above methods query information about the other side of the connection: EGGate, EGNode, the original DataFlowStar that the EGNode points to, and the invocation number of the EGNode.

`StringList printMe();`

It prints the information of the arc allocated: the sample rate and the initial delay.

`void setProperty(PortHole*` *`pPort`*, int *`index`*);

This method sets the pointer to the corresponding porthole of the original SDF graph and the index of the EGGate. Since multiple EGGates in an EGNode may be mapped to the same porthole in the original SDF graph, we order the EGGates by indices.

`void setLink(EGGateLink*` *`p`*);
`EGGateLink* getLink();`

Since the list of EGGates is maintained as a derived class of DoubleLinkList, an EGGate is assigned an EGGateLink that is derived from the DoubleLink class. These methods set and get the assigned EGGateLink.

`void hideMe(int` *`flag`*);

If the initial delay is greater than or equal to the sample rate in an EGArc, the precedence relationship between the source and the destination of the arc disappears while not removing the arc from the APEG. This method removes this EGGate from the access list of EGGates (`ancestors` or `descendants`), and stores it in the list of hidden EGGates (`hiddenGates`) of the parent EGNode . If the argument flag is NULL, it calls the same method for the EGGate of the other side of connection. By default, the flag is NULL.

`virtual ~EGGate();`

Is a virtual destructor that deletes the allocated arc, removes itself from the list of EGGates.

14.2.2 Class EGGateList

This class, derived from DoubleLinkList, contains a list of EGGates. An EGGate is assigned to an EGGateLink and the EGGateList class accesses an EGGate through the assigned EGGateLink.

The following ordering is maintained in the precedence list: entries for the same far-end EGNode occur together (one after another), and they occur in order of increasing invocation number. Entries for the same invocation occur in increasing order of the number of delays on the arc.

class EGGateLink

`EGGateLink(EGGate*` *`e`*`);`

The constructor has an argument for an EGGate.

```
EGGate* gate();
EGGateLink* nextLink();
```

These methods return the corresponding EGGate and the next link in the parent list.

`void removeMeFromList();`

Removes this link from the parent list.

EGGateList public members

Class EGGateList has a default constructor.

`void initialize();`

This method deletes all EGGates in the list and initialize the list. It is called inside the destructor.

`DoubleLink* createLink(EGGate*` *`e`*`);`

Creates an EGGateLink for the argument EGGate.

`void insertGate(EGGate*` *`e`*`, int` *`update`*`);`

This method insert a new EGGate into the proper position in the precedence list. The update parameter indicates whether or not to update the arc data if an EGGate with the same far-end EGNode and delay, already exists. If *`update`* is 0, the argument EGGate will be deleted if redundant. If 1, the arc information of the existing EGGate will be updated (sample rate will be increased). When we insert an EGGate to the `descendants` list of the parent EGNode, we set *`update`* to be 1. If the EGGate will be added to the `ancestors`, the variable is set 0.

`StringList printMe();`

Prints the list of EGGates.

Iterator for EGGateList

Class EGGateLinkIter is derived from class DoubleLinkIter. The constructor has an argument of the reference to a constant EGGateList object. It returns EGGates. This class has a special method to return the next EGGate connected to a new `farEndMaster` that is different from the argument DataFlowStar.

`EGGate* nextMaster(DataFlowStar*` *`master`*`);`

14.3 Class EGNode

Class EGNode is a node in an APEG, corresponding to an invocation of a DataFlowStar in the original SDF graph. The constructor has two arguments: the first argument is the pointer to the original Star of which it is an invocation, and the second argument represents the invocation number. The default value for the invocation number is 1. It has a virtual destructor that does nothing in this class.

An EGNode maintains three public lists of EGGates: `ancestors`, `descendants`, and `hiddenGates` .

14.3.1 Other EGNode public members

Invocations of the same DataFlowStar are linked together.

```
void setNextInvoc(EGNode* next);
EGNode* getNextInvoc();
EGNode* getInvocation(int i);
void setInvocationNumber(int i);
int invocationNumber();
```

The first two methods sets and gets the next invocation EGNode. The third method searches through the linked list starting from the current EGNode to return the invocation with the argument invocation number. If the argument is less than the invocation number of the current EGNode, returns 0. The other methods sets and gets the invocation number of the current EGNode.

`void deleteInvocChain();`

Deletes all EGNodes linked together starting from the current EGNode. This method is usually called at the EGNode of the first invocation.

```
StringList printMe();
StringList printShort();
```

These methods print the name and the invocation number. In the first method, the `ances-tors` and `descendants` lists are also printed.

`DataFlowStar* myMaster();`

Returns the original DataFlowStar of which the current EGNode is an invocation.

`int root();`

This method returns TRUE or FALSE, based on whether this node is a root of the APEG. A node is a root if it either has no ancestors, or if each arc in the ancestor list has enough delay on it.

`EGGate* makeArc(EGNode*` *`dest`*, int *`samples`*, int *`delay`*);

Create a connection from this node to the first argument node. A pair of EGGates and an EGArc are allocated in this method. This EGNode is assumed to be the source of the connection.

```
void resetVisit();
void beingVisited();
int alreadyVisited();
```

The above methods manipulates a flag for traversal algorithms: resets to 0, sets to 1, or queries the flag.

```
void claimSticky();
int sticky();
```

These methods manipulates another flag to indicate that the invocations of the same DataFlowStar may not be scheduled into different processors since there is a strong interdependency between them. The first method sets the flag and the second queries the flag.

14.3.2 EGNodeList

Class EGNodeList is derived from class DoubleLinkList.

`void append(EGNode*` *`node`*);
`void insert(EGNode*` *`node`*);

These methods appends or inserts the argument EGNode to the list.

```
EGNode* takeFromFront();
EGNode* headNode();
```

The above methods both returns the first EGNode in the list. The first method removes the node from the list while the second method does not.

There is a iterator class for the EGNodeList class, called EGNodeListIter. It returns the EGNodes.

14.4 Class ExpandedGraph

Class ExpandedGraph has a constructor with no argument and a virtual destructor that deletes all EGNodes in the graph.

The major method to generate an APEG is

`virtual int createMe(Galaxy&` *`galaxy`*, int *`selfLoopFlag`*);

The first argument is the original SDF galaxy of which the pointer will be stored in a protected member `myGal.` The second argument enforces to make arcs between invocations of the same star regardless of the dependency. The procedure of APEG generation is as follows.

5. Initialize the APEG graph.

`virtual void initialize();`

Does nothing here, but will be redefined in the derived class if necessary.

6. Allocate all invocations (EGNodes) of the blocks in the original SDF graph. Keep the list of the first invocations of all blocks in the protected member `masters.`

`virtual EGNode *newNode(DataFlowStar*` *`star`*, int *`invoc_index`*);

Is used to create an invocation of a DataFlowStar given as the first argument. The second argument is the invocation number of the node. This method is virtual since the derived ExpandedGraph class may have derived classes from the EGNode class.

7. For each star in the original SDF graph,

(3-1) Make connections between invocations of the star if any one of the conditions is met: *`selfLoopFlag`* is set in the second argument, the star has internal states, the star accesses past values on its portholes, or the star is a wormhole. The connection made in this stage does not indicate the flow path of samples, but the precedence relation of two EGNodes. Therefore, EGGates associated with this connection are not associated with portholes in the original SDF graph. If the connections are made, the `claimSticky` method of EGNode class is called for each invocation EGNode. If any such connection is made, the APEG is said not-parallelizable as a whole: A protected member, `parallelizable,` is set FALSE.

(3-2) For each input porthole, get the far-side output porthole and make connections between invocations of two DataFlowStars. A connection in the original SDF graph may be mapped to several connections in the APEG since the APEG is homogeneous.

8. Find the root nodes in the APEG and stored in its protected member `sources.`

`void insertSource(EGNode*` *`node`*);

Inserts the argument EGNode into the source list, `sources,` of the graph.

All protected members are explained above.

14.4.1 Other ExpandedGraph public members

`int numNodes();`

This method returns the number of total nodes in the APEG.

`virtual StringList display();`

Displays all EGNodes by calling `printMe` method of EGNode class.

`virtual void removeArcsWithDelay();`

This method hide all connections that have delays on them. When an APEG is created, the number of initial delays on an arc, if exists, is always greater than or equal to the sample rate of the arc. Therefore, this method is used to make the APEG actually acyclic.

14.4.2 Iterators for ExpandedGraph

There are three types of iterators associated with an ExpandedGraph: EGMasterIter, EGSourceIter, and EGIter. As its name suggests, EGMasterIter returns the EGNodes in `masters` list of the graph. EGSourceIter returns the EGNodes in `sources` list of the graph. Finally, EGIter returns all EGNodes of the ExpandedGraph.

EGMasterIter and EGSourceIter are derived from EGNodeListIter. EGIter, however, is not derived from any class. Instead, EGIter uses EGMasterIter to get the first invocation of each DataFlowStar in the original SDF graph and traverse the linked list of invocations. Thus invocations are traversed master by master.

Chapter 15. Parallel Schedulers

Authors: Soonhoi Ha

Base classes for parallel schedulers can be found in `$PTOLEMY/src/domains/cg/parScheduler`. All parallel schedulers use an APEG as the input. The APEG for parallel schedulers is called ParGraph, which is derived from class ExpandedGraph. Class ParNode, derived from class EGNode, is a node in a ParGraph.

The base scheduler object is ParScheduler. Since it is derived from class SDFScheduler, it inherits many methods and members from the SDFScheduler class. The ParScheduler class has a ParProcessors class that has member methods to implement the main scheduling algorithm. The ParProcessors class has an array of UniProcessor classes. The UniProcessor class, privately derived from class DoubleLinkList, is mapped to a processing element in the target architecture.

Note that all parallel scheduling algorithms are retargettable: they do not assume any specific topology while they take the effect of topology into account to estimate the interprocessor communication overhead.

Refer to class ParScheduler, to see the overall procedure of parallel scheduling. Refer to class UniProcessor, to see the procedure of sub-universe generations.

15.1 ParNode

This class represents a node in the APEG for parallel schedulers, thus contains additional members for parallel scheduling besides what are inherited from class EGNode. It has the same two-argument constructor as class EGNode.

`ParNode(DataFlowStar*` *`Mas`*`, int` *`invoc_no`*`);`

Initializes data members. If the argument star is at the wormhole boundary, we do not parallelize the invocations. Therefore, we create precedence relations between invocations by calling `claimSticky` of EGNode class in the constructor. If this constructor is called, the `type` protected member is set 0.

The ParNode class has another constructor with one argument.

`ParNode(int` *`type`*`);`

The scheduling result is stored in UniProcessor class as a list of ParNodes. This constructor is to model idle time (*`type`* = 1), or communication time (*`type`* = -1 for sending time, *`type`* = -2 for receiving time) as a ParNode. It initializes data members.

15.1.1 ParNode protected members

`int StaticLevel;`

Is set to the longest execution path to a termination node in the APEG. It defines the static level (or priority) of the node in Hu's scheduling algorithm. Initially it is set to 0.

`int procId`

Is the processor index on which this ParNode is scheduled. Initially it is set to 0.

```
int scheduledTime;
int finishTime;
```

Indicate when the node is scheduled and finished, respectively.

`int exTime;`

Is the execution time of the node. If it is a regular node (*`type`* = 0), it is set to the execution time of the original DataFlowStar. Otherwise, it is set to 0.

`int waitNum;`

Indicates the number of ancestors to be scheduled before scheduling this node. during the scheduling procedure. Initially it is set to 0. At a certain point of scheduling procedure, we can schedule a ParNode only when all ancestors are already assigned, or `waitNum` is 0.

```
EGNodeList tempAncs;
EGNodeList tempDescs;
```

These list members are copies of the ancestors and descendants of the node. While EGGateLists, `ancestors` and `descendants,` may not be modified during scheduling procedure, these lists can be modified.

15.1.2 Other ParNode public members

```
void assignSL(int SL);
int getSL();
virtual int getLevel();
```

The first two methods set and get the `StaticLevel` member. The last one returns the priority of the node, which is just `StaticLevel` by default. In the derived classes, this method can be redefined, for example in Dynamic Level Scheduling, to return the dynamic level of the node.

`int getType();`

Returns the type of the node.

```
void setProcId(int i);
int getProcId();
```

These two methods set and get the `procId` member.

```
void setScheduledTime(int i);
int getScheduledTime();
void setFinishTime(int i);
int getFinishTime();
```

These methods are used to set or get the time when the node is scheduled first and finished.

```
void setExTime(int i);
int getExTime();
```

These methods are used to set and get the execution time of the node.

`void resetWaitNum();`
`void incWaitNum();`

Resets the `waitNum` variable to the number of ancestors, and increases it by 1.

`int fireable();`

This method decreases `waitNum` by one, and return `TRUE` or `FALSE`, based on whether `waitNum` reaches zero or not. If it reaches 0, the node is declared "fireable".

`void copyAncDesc(ParGraph*` *g*, int *flag*);
`void removeDescs(ParNode*` *n*);
`void removeAncs(ParNode*` *n*);
`void connectedTo(ParNode*` *to*);

The first method initializes the lists of temporary ancestors and descendants, `tempAncs` and `tempDescs,` from `ancestors` and `descendants` that are inherited members from EGNode class. List `tempAncs` is sorted smallest `StaticLevel` first while list `tempDescs` is sorted largest `StaticLevel` first. The first argument is necessary to call the sorting routine which is defined in the ParGraph class . By virtue of sorting, we can traverse descendant with larger `StaticLevel` first. If the second argument is not 0, we switch the lists: copy `ancestors` to `tempDescs` and `descendants` to `tempAncs.`

The second and the third methods remove the argument node from the temporary descendant list or from the temporary ancestor list. In the latter case, we decrease `waitNum` by one.

The last method above is to make a temporary connection between the node as the source and the argument node as the destination. The temporary descendant list of the current node is added the argument node while the temporary ancestor list of the argument node is added the current node (also increase `waitNum` of the argument node by 1).

`CGStar* myStar();`

Returns the original DataFlowStar after casting the type to CGStar, star class type of the CG domain.

`int amIBig();`

Returns `TRUE` or `FALSE`, based on whether `myStar` is a wormhole or not. Before the scheduling is performed in the top-level graph, the wormhole executes scheduling the inside galaxy and stores the scheduling results in the Profile object . The ParNode keeps the pointer to the Profile object if it is an invocation of the wormhole. In the general context, the node will be considered "Big" if the master star can be scheduled onto more than one processors. Then, the star is supposed to keep the Profile object to store the schedules on the processors. A wormhole is a special case of those masters.

`void setOSOPflag(int` *i*);
`int isOSOP();`

After scheduling is performed, we set a flag to indicate whether all invocations of a star are assigned to the same processor or not, using the first method. The second method queries the flag. Note that only the *first* invocation has the valid information.

```
void setCopyStar(DataFlowStar* s, ParNode* prevN);
DataFlowStar* getCopyStar();
ParNode* getNextNode();
ParNode* getFirstNode();
int numAssigned();
```

The above methods are used to create sub-universes . When we create a sub-universe, we make a copy of the master star if some invocations are assigned to the processor. Then, these invocations keep the pointer to the cloned star. Since all invocations may not be assigned to the same processor, we maintain the list of invocations assigned to the given processor. The first and second methods set and get the pointer to the cloned star. The first method also make a chain of the invocations assigned to the same processor. The third method returns the next invocation chained from the current node, while the fourth method returns the starting invocation of the chain. The last method returns the total number of invocations in the chain. It should be called at the starting invocation of the chain.

```
void setOrigin(EGGate* g);
EGGate* getOrigin();
void setPartner(ParNode* n);
ParNode* getPartner();
```

These methods manipulate the connection information of communication nodes. If two adjacent nodes in an APEG are assigned to two different processors, we insert communication nodes between them: Send and Receive nodes. As explained earlier, a communication node is created by one-argument constructor. The first two methods are related to which EGGate the communication node is connected. The last two methods concern the other communication node inserted.

15.1.3 Iterators for ParNode

There are two types of iterators associated with ParNode class: ParAncestorIter, ParDescendantIter. As their names suggest, ParAncestorIter class returns the ParNodes in the temporary ancestor list (`tempAncs`), and ParDescendantIter class returns the ParNodes in the temporary descendant list (`tempDescs`).

15.2 Class ParGraph

Class ParGraph, derived from class ExpandedGraph, is an APEG graph for parallel schedulers. It has a constructor with no argument.

```
int createMe(Galaxy& g, int selfLoopFlag = 0);
```

Is the main routine to create and initialize the APEG of the argument Galaxy. Using `createMe` method of the ExpandedGraph class, it creates an APEG. After that, it resets the busy flags of the ParNodes, and calls

```
virtual int initializeGraph();
```

This is a protected method. It performs 4 main tasks as follows. (1) Call a protected method `removeArcsWithDelay` to remove the arcs with delays, and to store the source and the destination nodes of each removed arc into the list of NodePairs . The list is a protected member, named `nodePairs` of SequentialList class.

`void removeArcsWithDelay();`
`SequentialList nodePairs;`

(2) For each node, compute the static level (`StaticLevel`) by calling a protected method `SetNodeSL.`

`int SetNodeSL(ParNode*` *n*);

(3) Sum the execution times of all nodes and save the total execution time to a protected member `ExecTotal.`

`int ExecTotal;`

(4) Assign the larger static level than any other nodes to the nodes at the wormhole boundary. This let the parallel scheduler schedules the nodes at the wormhole boundary first.

15.2.1 Other ParGraph protected members

`EGNode* newNode(DataFlowStar*, int);`

Redefines the virtual method to create a ParNode associated with the given invocation of the argument star.

`ostream* logstrm;`

This is a stream object for logging information.

15.2.2 Other ParGraph public members

`EGNodeList runnableNodes;`
`void findRunnableNodes();`

The list of runnable (or fireable) nodes are stored in `runnableNodes.` The above method is to initialize the list with all root ParNodes.

`int getExecTotal();`
`Galaxy* myGalaxy();`

Returns the total execution time of the graph and the original graph.

`void setLog(ostream*` *l*);

Sets the stream object `logstrm.`

`void replenish(int` *flag*);

This method initialize the temporary ancestor list and descendant list of all ParNodes in the graph.

`void sortedInsert(EGNodeList&` *l*, ParNode* *n*, int *flag*);

Insert a ParNode, *n,* into the EGNodeList, *l,* in sorted order. It sorts nodes of highest `StaticLevel` first if *flag* = 1, or lowest `StaticLevel` first if *flag* = 0.

`void restoreHiddenGates();`

This method restores the hidden EGGates from `removeArcsWithDelay` method to the initial list, either `ancestors` or `descendants` of the parent node.

`int pairDistance();`

After scheduling is completed, it is supposed to return the maximum scheduling distance between node pairs in `nodePairs` list. Currently, however, it just returns -1, indicating the information is not available.

`~ParGraph();`

The destructor initializes the `nodePairs` list.

15.2.3 Class NodePair

Class NodePair saves the source and the destination ParNodes of an arc.

`NodePair(ParNode*` *`src`*, ParNode* *`dest`*);
`ParNode* getStart();`
`ParNode* getDest();`

The constructor requires two arguments of the source and the destination nodes, while the next two methods return the node.

15.3 Class ParScheduler

Class ParScheduler is derived from class SDFScheduler, thus inherits the most parts of `setup` method. They include initialization of galaxy and computation of the repetition counters of all stars in the SDF graph. It redefines the scheduling part of the set-up stage (`computeSched-ule).`

`int computeSchedule(Galaxy&` *`g`*);

Is a protected method to schedule the graph with given number of processors. The procedure is

(1) Let the target class do preparation steps if necessary before scheduling begins.

(2) Check whether the number of processors is 1 or not. If it is 1, we use the single processor scheduling (`SDFScheduler` :: computeSchedule). After we set the target pointer of each star, return.

(3) Form the APEG of the argument galaxy, and set the total execution time of the graph to a protected member `totalWork.`

(4) Set the target pointer of each UniProcessor class .

void mapTargets(IntArray* *array* = 0);

If no argument is given, assign the child targets to the UniProcessors sequentially. If the IntArray argument maps the child targets to the UniProcessors. If array[1] = 2, UniProcessor 1 is assigned Target 2.

(5) Before the main scheduling begins, complete the profile information of wormholes. Since we may want to perform more tasks before scheduling, make this protected method virtual. Be default, return `TRUE` to indicate no error occurs.

virtual int preSchedule();

(6) Perform scheduling by calling `mainSchedule.`

`int mainSchedule();`

This public method first checks whether manual assignment is requested or not. If it is, do manual assignment. Otherwise, call an automatic scheduling routine which will be redefined in each derived class, actual scheduling class. After scheduling is performed, set the `procId` parameter of the stars in the original galaxy if all invocations are enforced to be assigned to the same processor .

`int assignManually();`

Is a protected method to return `TRUE` if manual assignment is requested, or return `FALSE` otherwise.

`virtual int scheduleManually();`

Is a public virtual method. This method first checks whether all stars are assigned to processors (`procId` parameter of a star should be non-negative and smaller than the number of processors). If there is any star unassigned, return `FALSE`. All invocations of a star is set the same `procId` parameter. Based on that assignment, perform the list scheduling . The `procId` of a Fork star is determined by its ancestor. If the ancestor is a wormhole, the `procId` of the Fork should be given explicitly as other stars.

`virtual int scheduleIt();`

Is a public virtual method for automatic scheduling. Refer to the derived schedulers. By default, it does nothing and return `FALSE` to indicate that the actual scheduling is not done in this class.

`int OSOPreq();`

Is a protected method to return `TRUE` or `FALSE`, based on whether all invocations are enforced to be scheduled on the same processor.

Now, all methods necessary for step (5) are explained. Go back to the next step.

(7) As the final step, we schedule the inside of wormholes based on the main scheduling result if automatic scheduling option is taken. In the main scheduling routine, we will determine how many processors will be assigned to a wormhole.

`int finalSchedule();`

If scheduling of wormholes succeeds, return `TRUE`. Otherwise, return `FALSE`.

15.3.1 compileRun method

`void compileRun();`

Is a redefined public method of SDFScheduler class. It first checks the number of processors. If the number is 0, it just calls `SDFScheduler` :: compileRun. This case occurs inside a wormhole. Otherwise,

(1) Set the target pointer of UniProcessors.

(2) Create sub-universes for each processors.

`int createSubGals(Galaxy&` *g*);

Is a public method. It first checks whether all invocations of stars are scheduled on the same processor, and set the flag if it is the case. After restoring all hidden EGGates of the APEG, create sub-universes.

(3) Prepare each processor (or UniProcessor class) for code generation. It includes sub-universe initialization, and simulation of the schedule on the processor obtained from the parallel scheduling.

(4) Let the target do something necessary, if any, before generating code.

(5) Generate code for all processors.

15.3.2 Other ParScheduler protected members

```
const char* logFile;
pt_ofstream logstrm_real;
ostream* logstrm;
```

These are for logging information. `logFile` indicates where to store the logging information.

```
MultiTarget* mtarget;
```

Is the pointer to the target object, which is MultiTarget type.

```
int numProcs;
```

Is the total number of processors.

```
ParGraph *exGraph
```

Is the pointer to the APEG used as the input graph to the scheduler.

```
ParProcessors* parProcs;
```

This member points the actual scheduler object. It will be set up in the `setUpProcs` method of the derived class.

```
IntArray avail;
```

This array is to monitor the pattern of processor availability during scheduling.

```
int inUniv;
```

This flag is set `TRUE` when it is the scheduler of a universe, not a wormhole. In the latter case, it is set `FALSE`. By default, it is set `TRUE`.

```
int withParallelStar();
```

This method returns `TRUE` or `FALSE`, based on whether the galaxy contains any wormhole or data-parallel star, or not.

```
int overrideSchedule();
```

If the user wants to override the scheduling result after automatic scheduling, he can set the `adjustSchedule` parameter of the target object. This method pokes the value of that parameter. This is one of the future feature, not implemented yet in Ptolemy due to limitation of the graphical interface, pigi.

15.3.3 Other ParScheduler public members

`ParScheduler(MultiTarget*` *`t`*, const char* *`log`* = 0);
`virtual ~ParScheduler();`

The constructor has two arguments: the target pointer and the name of log file name. The virtual destructor does nothing.

`virtual void setUpProcs(int` *`num`*`);`

The number of processors is given as an argument to this method. It will initialize the `avail` array. In the derived class, this method will create a ParProcessors class (set `parProcs` member).

`ParProcessors* myProcs();`
`UniProcessor* getProc(int` *`n`*`);`

These methods will return the pointer to the ParProcessors object associated with this scheduler and the UniProcessor object indexed by the argument. The range of the index is 0 to `numProcs-1.`

`void ofWorm();`

Resets `inUniv` flag to `FALSE`.

`int getTotalWork();`

Returns the total execution time of the graph.

`void setProfile(Profile*` *`profile`*`);`

Copy the scheduling results to the argument Profile . If the scheduling is inside a wormhole, the scheduling results should be passed to the outside of the wormhole by a Profile object.

15.4 class ParProcessors

Class ParProcessors is the base class for all actual scheduler object. Refer to derived classes to see how scheduling is performed. This class just provide the set of common members and methods. Among them, there is a list scheduling routine.

`int listSchedule(ParGraph*` *`graph`*`);`

This method performs the list scheduling with the input argument APEG. It should be called after all nodes are assigned to the processors. It is the last routine to be called for all parallel schedulers. It adds communication nodes to the APEG `(findCommNodes)` and schedule them with the regular ParNodes. It returns the makespan of the schedule.

`void findCommNodes(ParGraph*` *`graph`*`);`

This method puts a pair of communication ParNodes, a send node and a receive node, on the arc between two nodes assigned to the different processors. Note that we use `tempAncs` and `tempDescs` list of ParNode class to insert these nodes instead of modifying the APEG. We store the newly created communication ParNodes in `SCommNodes.` The procedure consists of two stages. In the first stage, all regular arcs in the APEG are considered. The `StaticLevel` of the send node is assigned to that of the source node plus one to ensure that the send node is scheduled right after the source node. The static level of the receive node is assigned to the same value as the destination node. In the second stage, all hidden arcs are considered. In this case, the `StaticLevel` of communication nodes are assigned to 1, the minimum value since they may be scheduled at the end of the schedule. The number of interprocessor requirements are saved in a protected member, `com-`

`mCount.`

`int getMakespan();`

Returns the longest scheduled time among all UniProcessors.

15.4.1 Other ParProcessors protected members

```
int numProcs;
MultiTarget* mtarget;
EGNodeList SCommNodes;
```

These members specify the number of processors, the pointer to the multiprocessor target class, and the list of communication nodes added during `listSchedule.`

`IntArray pIndex;`

Is used to access the processors in the order of available time.

`void scheduleParNode(ParNode* `*`node`*`);`

This method schedules a parallel node (a wormhole or a data-parallel star) inside the `listSchedule` method. Note that the processors are already assigned for the node.

`virtual ParNode* createCommNode(int `*`i`*`);`

Is a virtual method to create a ParNode with type given as an argument. It is virtual since the derived scheduler may want to create a node of derived class of ParNode.

`void removeCommNodes();`

Clears the `SCommNodes` list.

`void sortWithAvailTime(int `*`guard`*`);`

Sort the processors with their available times unless no node is assigned to the processor. All idle processors are appended after the processors that are available at *`guard`* time and before the processor busy at *`guard`* time. Store the results to `pIndex` array.

`int OSOPreq();`

Returns `TRUE` or `FALSE`, based on whether all invocations of a star are enforced to be scheduled on the same processor or not.

15.4.2 Other ParProcessors public members

```
ParProcessors(int, MultiTarget*);
virtual ~ParProcessors();
```

The constructor has two arguments: the number of processors and the target pointer. It creates `pIndex` array and initialize other data structures. The destructor clears `SCommNodes.`

```
void mapTargets(IntArray* array);
void prepareCodeGen();
void createSubGals();
void generateCode();
```

The above methods perform the actual action defined in the ParScheduler class. For description, refer to class ParScheduler. The last method deliver the generate code from each processor to the target class.

`int size();`

returns the number of processors.

`virtual UniProcessor* getProc(int` *num*);

This method returns the UniProcessor with a given index. It is virtual since the derived class wants to return it own specific class derived from UniProcessor class.

`void initialize();`

Initializes `pIndex, SCommNodes,` and processors.

`StringList display(NamedObj*` *gal*);
`StringList displaySubUnivs();`

These methods return the StringList contains the scheduling result and the sub-universe description.

`ParNode* matchCommNodes(DataFlowStar*, EGGate*, PortHole);`

This method is used in sub-universe generation. The first argument is a communication star, either a send star or a receive star, that the system automatically inserts for interprocessor communication. The second argument is the EGGate that the interprocessor communication (IPC) occurs. If the second argument is `NULL`, the third argument indicates the porthole that the IPC occurs. In case all invocations of any star are assigned to the same processor, the sub-universe creation procedure is greatly simplified: we do not need to look at the APEG, rather look at the original SDF graph to create the sub-universe. It is the case when the second argument becomes `NULL`. This method sets the pointer of the communication star to the corresponding ParNode that are inserted during `listSchedule` method.

15.5 UniProcessor

Class UniProcessor simulates a single processing element, or a processor. It is derived from class DoubleLinkList to hold the list of ParNodes from parallel scheduling. Class NodeSchedule is derived from class DoubleLink to register a ParNode into the DoubleLinkList.

A UniProcessor keeps two target pointers: one for multiprocessor target (`mtarget),` and the other for the processor (`targetPtr).` They are both protected members.

```
MultiTarget* mtarget;
CGTarget* targetPtr;
```

The pointer to the processor can be obtained by a public method:

`CGTarget* target();`

The pointers to the multiprocessor target and to the ParProcessors class that this UniProcessor belongs to, are set by the following method:

`void setTarget(MultiTarget*` *t*, ParProcessors* *p*);

15.5.1 Class NodeSchedule

A NodeSchedule is an object to link a ParNode to a linked list. It indicates whether the node represents an idle time slot or not. It also contains the duration of the node. There is no protected member in this class.

```
void resetMembers();
void setMembers(ParNode* n, int dur);
```

These methods set the information for the associated node: the pointer to the node, duration, and a flag to tell whether it is an idle node or not. In the first method, the idle flag is set `FALSE`. The constructor also resets all internal information of the class.

```
ParNode getNode();
int getDuration();
int isIdleTime();
```

The above methods return the pointer to the node associated with this class, its duration, and the flag to say `TRUE` if it represents an idle time slot.

```
NodeSchedule* nextLink();
NodeSchedule* previousLink();
```

These methods return the next and the previous link in the linked list.

15.5.2 Members for scheduling

Since a list scheduling (with fixed assignment) will be performed as the last stage of all scheduling algorithms in Ptolemy , basic methods for list scheduling are defined in the UniProcessor class. In list scheduling, we need the available time of the processor.

```
int availTime;
```

Is a protected member to indicate the time when the processor available. There are public methods to access this member:

```
void setAvailTime(int t);
int getAvailTime();
NodeSchedule* curSchedule;
```

This protected member points to the NodeSchedule appended last to the linked list. There are two public methods to access a NodeSchedule:

```
NodeSchedule* getCurSchedule();
NodeSchedule* getNodeSchedule(ParNode* n);
```

The first method just returns `curSchedule` member while the second one returns the NodeSchedule associated with the argument ParNode.

When a ParNode is runnable earlier than the available time of the processor, we want to check whether there is an idle slot before `availTime` to fit the ParNode in the middle of the schedule:

```
int filledInIdleSlot(ParNode*, int start, int limit = 0);
```

The first two arguments are the pointer to the ParNode to be scheduled and the earliest time when the node can be scheduled. Without the third argument given explicitly, this method returns the earliest time that the processor is available to schedule the node. If the third argument is given, the available time of the processor should be less than *`limit`*. If this method could not find an idle slot to schedule the node, it returns -1. Otherwise, it returns the possible scheduling time of the node.

`int schedInMiddle(ParNode* n, int when, int);`

Schedule the node, *`n`*, at *`when`* inside an idle-time slot of the processor. The third argument indicates the duration of the node. This method returns the completion time of the schedule if scheduling is succeeded. If it fails to find an idle-time slot at *`when`* to accommodate the node, it returns -1.

If a node is to be appended at the end of the schedule in a processor,

`void appendNode(ParNode* n, int val);`

Does that blindly. To schedule a non-idle node, we have to use

`int schedAtEnd(ParNode* n, int start, int leng);`

In case *`start`* is larger than the processor available time, this method put an idle time slot in the processor and calls `appendNode`. And, it sets the schedule information of the node, and increases `availTime` of the processor.

`void addNode(ParNode* node, int start);`

This method is given a ParNode and its runnable time, and schedule the node either inside an idle time slot if possible, or at the end of the schedule list. The methods described above are used in this method.

`void scheduleCommNode(ParNode* n, int start);`

When we schedule the given communication node, *`n`*, available at *`start`*, we also have to check the communication resource whether the resources are available or not. For that purpose, we detect the time slot in this processor to schedule the node, and check whether the same time slot is available in the communication resources: we use `scheduleComm` of the multiprocessor target to check it. If we find a time slot available for this processor and the communication resources, we schedule the communication node.

`int getStartTime();`

Returns the earliest time when the processor is busy.

All methods described in this sub-section are public.

15.5.3 Sub-Universe creation

After scheduling is performed, a processor is given a set of assigned ParNodes. Using the scheduling result, we will generate code for the target processor. To generate code for the assigned nodes, we need to allocate resources for the nodes and examine the connectivity of the nodes in the original graph. These steps are common to the generic routine for single processor code generation in which a processor target is given a galaxy. Therefore, we want to create a sub-galaxy that consists of stars of which any invocation is assigned to the processor. Note that a sub-galaxy is NOT a subgraph of the original SDF graph. Besides the stars in the original program graph, we include other stars such as communication stars and spread/collect stars. This subsection will explain some details of sub-universe creation.

`void createSubGal();`

Is the main public method for sub-universe creation. It first creates a galaxy data structure, `subGal,` a private member. Then, it clones stars of at least one of whose invocations is assigned to the processor. Make a linked list for all assigned invocations (nodes) of each star in order of increasing invocation number, and set the pointer of cloned star. As for a wormhole, we create a CGWormStar instead of cloning the wormhole. A CGWormStar class will replace a wormhole in the sub-universe. If an original star is not supported by the processor target (for example, with heterogeneous scheduling), we create a star with the same name as the original star in the target domain.

In the next step, we connect the cloned stars by referring to the original galaxy. If a star is at the wormhole boundary in the original graph, we connect the cloned star to the same event horizon; by doing so the wormhole in the original graph is connected to the sub-universe. If the star at the wormhole boundary is scheduled on more than one processors (or not all invocations are assigned to the same processor), the wormhole in the original graph will be connected to the last created sub-universe.

If an arc connects two stars who have some invocations assigned to the same processor, we examine whether all of the two stars' invocations are assigned to the same processor. If they are, we just connect the cloned stars in the sub-universe. If they aren't, we have a cloned star of either one star whose invocations are assigned to the current sub-universe. In this case, we create a send star (`createSend` method of the multiprocessor target) or a receive star (`createReceive` of the target), based on whether the cloned star is a source or destination of the connection . We create a communication star and set the communication star pointer of the communication nodes in the APEG , by `matchCommNodes` method of ParProcessors class. If the partner communication star was already created in another sub-universe, we pair the send and receive stars by `pairSendReceive` method of the multiprocessor target .

The last case is when an arc connects two stars whose invocations are distributed over more than one processor. If no invocation of the destination star is assigned to this processor, we call the `makeBoundary` method.

`void makeBoundary(ParNode*` ***`src`***`, PortHole*` ***`orgP`***`);`

The first argument points to the earliest invocation of the star assigned to this processor, and the second is the pointer to the output porthole in the original SDF graph as the source of the connection. We examine all assigned invocations to check whether the destination invocations are assigned to the same processor. If they are, we create one send star and connect it to the cloned star. Otherwise, we create a Spread star to distribute the output samples to multiple processors. We connect the cloned star to the Spread star, and the Spread star to multiple send stars.

Otherwise, we call the `makeConnection` method.

`void makeConnection(ParNode*` ***`dest`***`, ParNode*` ***`src`***`, PortHole*` ***`ref`***`, ParNode*` ***`firstS`***`);`

The first argument is the pointer to the first assigned invocation of the destination star while the second one is the source node connected to the first argument. The third argument is the pointer to the destination porthole of the connection in the original graph. The last argument is the pointer to the first invocation of the source star. Note that the last argument node may not be assigned to the current processor. This method examines all assigned invocations of the destination star to identify the sources of the samples to be consumed by the cloned star in this processor. If the number of sources are more than one, we create a Collect star and connect the Collect star to the cloned destination star in the sub-universe. For each source in the other processors, we create a receive star and connect it to the Collect star. Similarly, we examine all assigned invocations of the source star to identify the destinations of the samples to be produced by the source star in this processor. If the number of destinations is more than one, we create a Spread star and connect it to the source star. For each destination in the other processors, we create a send star and connect it to the Spread star. As a result, it may occur that to connect two stars in the sub-universe, we need to splice a Spread and a Collect star on that connection.

Spread and Collect stars

A Spread or a Collect star is created by `createSpread` or `createCollect` method of the multiprocessor target. The following illustrates when we need to create a Spread or a Collect star in a sub-universe.

Suppose we have star A connected to star B in original graph. Star A produces two samples and star B consumes one. Then, one invocation of star A is connected to two invocations of star B. If one invocation of star A and only one of invocation of star B are assigned to the current processor. Then, we need to connect the cloned stars of A and B in the sub-universe. We can not connect stars A and B in the sub-universe directly since among two samples generated by star A, one sample should be transferred to another processor through a send star. In this case, we connect a Spread star to star A, and one send star and star B to the Spread star in the sub-universe. Then, star A produces two samples to the Spread star while the Spread star *spread* the incoming two samples to the send star and star B one sample each. The number of output portholes and the sample rate of each porthole are determined during the sub-universe creation. If there is no initial delay on the connection and neither star A nor B needs to access the past samples, the Spread star does not imply additional copying of data in the generated code.

Similarly, we need to connect a Collect star to the destination star if samples to that star come from more than one sources.

15.5.4 Members for code generation

`void prepareCodeGen();`

This method performs the following tasks before generating code.

(1) Initialize the sub-universe, which also initialize the cloned stars.

(2) Convert a schedule (or a linked list of ParNodes) obtained from the parallel scheduler to a SDFSchedule class format (list of stars). The code generation routine of the processor target assumes the SDFSchedule class as the schedule output.

(3) Simulate the schedule to compute the maximum number of samples to be collected

at runtime if we follow the schedule. This information is used to determine the buffer size to be allocated to the arcs.

`void simRunSchedule();`

Performs the step (3). It is a protected member.

`StringList& generateCode();`

This method generate code for the processor target by calling `targetPtr->generateCode()`.

`int genCodeTo(Target*` *t*);

This method is used to insert the code of the sub-universe to the argument target class. It performs the almost same steps as `prepareCodeGen` and then calls `insertGalaxyCode` of the processor target class instead of `generateCode` method.

15.5.5 Other UniProcessor protected members

There are a set of methods to manage NodeSchedule objects to minimize the runtime memory usage as well as execution time.

`void putFree(NodeSchedule*` *n*);
`NodeSchedule* getFree();`
`void clearFree();`

If a NodeSchedule is removed from the list, it is put into a pool of NodeSchedule objects. When we need a NodeSchedule object, a NodeSchedule in the pool is extracted and initialized. We deallocate all NodeSchedules in the pool by the third method.

`void removeLink(NodeSchedule*` *x*);

Removes the argument NodeSchedule from the scheduled list.

`int sumIdle;`

Indicates the sum of the idle time slots after scheduling is completed. The value is valid only after `display` method is called.

15.5.6 Other UniProcessor public members

There are a constructor with no argument to initialize all data members and a destructor to delete `subGal` and to delete all NodeSchedule objects associated with this processor.

```
Galaxy* myGalaxy();
int myId();
DoubleLinkList :: size;
int getSumIdle();
```

The above methods return the pointer to the sub-universe, the index of the current processor, the number of scheduled nodes, and the sum of idle time after scheduling is completed (or `sumIdle`).

`void initialize();`

This method puts all NodeSchedules in the list to the pool of free NodeSchedules and initialize protected members.

`void copy(UniProcessor* org);`

Copies the schedule from the argument UniProcessor to the current processor.

```
StringList displaySubUniv();
StringList display(int makespan);
int writeGantt(ostream& os, const char* universe, int numProcs, int
span);
```

The first method display the sub-universe. The second method displays the schedule in the textual form while the third one forms a string to be used by Gantt-chart display program.

15.5.7 Iterator for UniProcessor

Class ProcessorIter is the iterator for UniProcessor class. A NodeSchedule object is returned by `next` and ++ operator.

`ParNode* nextNode();`

Returns the ParNode in the list.

15.6 Dynamic Level Scheduler

Dynamic Level Scheduling is one of the list scheduling algorithms where the priority of a node is not fixed during the scheduling procedure. The scheduling algorithm is implemented in `$PTOLEMY/src/domains/cg/dlScheduler`. All classes in that directory are derived from the base parallel scheduling classes described above in this chapter. For example, DLNode class is derived from class ParNode, and redefines `getLevel` method to compute the *dynamic* level of the node.

`int getLevel();`

This method returns the sum of the static node and the worst case communication cost between its ancestors and this DLNode.

Class DLNode has the same constructors as class ParNode.

The dynamic level scheduler maintains a list of runnable nodes sorted by the `getLevel` value of the DLNodes. It fetches a node of highest priority and choose the best processor that can schedule the node earliest while taking interprocessor communication into account.

15.7 Class DLGraph

Class DLGraph, derived from class ParGraph, is the input APEG graph to the dynamic level scheduler. It consists of DLNode objects created by redefining the following method:

`EGNode* newNode(DataFlowStar* s, int i);`

This method creates a node in the APEG graph. Here, it creates a DLNode.

DLGraph has a protected member maintaining the number of unscheduled nodes.

`int unschedNodes;`

We may check whether the scheduler is deadlocked or not by examining this variable when the scheduler halts. This can be manipulated by the public methods

```
void decreaseNodes();
int numUnSchedNodes();
```

The first method decrements `unschedNodes` and the second method returns it.

The DLGraph class redefines `resetGraph` method.

```
void resetGraph();
```

This makes the initial list of runnable nodes and sets the variable described above. This method internally calls the following protected method:

```
virtual void resetNodes();
```

This method resets the busy flag and the `waitNum` member of DLNodes.

There are three other public members.

```
DLNode* fetchNode();
```

Fetches a DLNode from the head of the list of runnable nodes.

```
StringList display();
```

Displays the APEG and the list of source nodes.

15.8 class DLScheduler

Class DLScheduler is derived from class ParScheduler. It has a constructor with three arguments.

```
DLScheduler(MultiTarget* t, const char* log, int i);
```

The arguments are the pointer to the multiprocessor target, the name of the logging file, and a flag to indicate whether the communication overhead can be ignored or not. If the processor target has special hardware for communication separate from the CPU, then most of the communication can be simultaneous with processor computation. In this case, we do not reserve the communication time slot in the processor schedule but in the access schedule of the communication resources only. This mode of operation is selecte if *`i`* is set `TRUE`. This is not implemented since we haven't dealt with that kind of architecture yet.

There is a protected member to point to a ParProcessor class:

DLParProcs* parSched;

This pointer is set in the following redefined method:

```
void setUpProcs(int num);
```

This method first performs `ParScheduler::setUpProcs`, and then create a DLParProcs object. While this class defines the overall procedure of the dynamic level scheduling algorithm, the DLParProcs class provides the details of the algorithm .

```
~DLScheduler();
```

Deallocate the DLParProcs object.

The main procedure of the dynamic level schedule is defined in
`int scheduleIt();`
This method does the following:

(1) Initializes the DLParProcs and resets the DLGraph and the communication resources of the multiprocessor target.

(2) Fetch a node from the list of runnable nodes until there are no more nodes in the list.

(2-1) If the node is not a parallel node, call `scheduleSmall` method of the DLParProcs class to schedule the node.

(2-2) If the node is the first invocation of a parallel-star node, we give up. NOTE: We do not support parallel stars since we haven't had to deal with them yet.

(3) When there are no more runnable nodes, we check whether the graph is deadlocked. In case of successful completion, we perform an additional list scheduling (`listSchedule` of the ParProcessors class), based on the processor assignment determined by the above procedure.
`StringList displaySchedule();`
Displays the final schedule results.

15.9 Class DLParProcs

Class DLParProcs, derived from the ParProcessors class, defines a main object to perform the dynamic level scheduling algorithm. It has a constructor with two arguments:

`DLParProcs(int` *pNum*, `MultiTarget*` *t*);
The arguments are the number of processors and the pointer to the multiprocessor target. This method creates *pNum* UniProcessors for processing elements. These UniProcessors are deallocated in the destructor `~DLParProcs().`

Based on the type of node described in `scheduleIt` method of the DLScheduler class, we call one of the following methods to schedule the node. `scheduleSmall, scheduleBig, copyBigSchedule.`

`virtual void scheduleSmall(DLNode*` *n*);
This method is a public method to schedule an atomic node that will be scheduled on one processor. This is virtual since the HuParProcs class redefines this method. The scheduling procedure is as follows:

(1) Obtain the list of processors that can schedule this node. Refer to `candidateProcs` method of the CGMultiTarget class . Here, we examine the resource restriction of the processor, as well as the types of stars that a processor supports, in case of a heterogeneous target. If all invocations of a star should be scheduled to the same processor and another invocation of the star is already scheduled, we put only that processor only into the list of candidate processors that can schedule this node.

(2) Among all candidate processors, we select the processor that can schedule this node the earliest. In this stage, we consider all communication overhead if ancestors would be

assigned to the different processors.

(3) Assign the node to that processor:

`void assignNode(DLNode* n, int destP, int time);`

Is a protected method to assign an atomic node (`n`) to the processor of index `destP` at `time.` This method also schedules the communication requirements in the communication resources.

(4) Indicate that the node was fired:

`virtual void fireNode(DLNode* node);`

It is a virtual and protected method. It fires the argument node and insert its descendants into the list of runnable nodes if they become runnable after this node is fired.

(5) Finally, we decrease the number of unscheduled nodes and the total remaining work of the DLGraph class.

The DLNode class has the pointer to the Profile class that determines the inside schedule of the node. We insert idle time slots to the processors to match the pattern of processor availability with the starting pattern of the profile and append the node at the end of the processors. We record the assignment of the profile to the processors in `assignedId` array of the Profile class . We also save the scheduling information in the DLNode: when the node is scheduled and completed, and on which processor it is scheduled. To determine the latter, we select the processor that the inside scheduler assumes the first processor it is assigned.

After appending the profile at the end of the available processors, we fire the node and update the variables of the DLGraph.

`void initialize(DLGraph* graph);`

This method calls `ParProcessors` :: initialize and resets the `candidate` member to the index of the first UniProcessor, and `myGraph` member to `graph` argument.

`DLGraph* myGraph;`

A protected members to store the pointer to the input APEG.

`int costAssignedTo(DLNode* node, int destP, int start);`

This method is a protected method to compute the earliest time when the processor of index `destP` could schedule the node `node` that is runnable at time `start.`

15.10 Hu Level Scheduler

Hu’s level scheduling algorithm is a simple list scheduling algorithm, in which a node is assigned a fixed priority. No communication overhead is considered in the scheduling procedure. The code lies in `$PTOLEMY/src/domains/cg/HuScheduler`. All classes, except the HuScheduler class in this directory, are derived from the classes for the dynamic level schedulers.

15.10.1 Class HuNode

Class HuNode represents a node in the APEG for Hu’s level scheduling algorithm. It is derived from the DLNode class so that it has the same constructors. The level (or priority) of a node does not depend on the communication overhead.

`int getLevel();`

Just returns `StaticLevel` of the node.

A HuNode has two private variables to indicate the available time of the node (or the time the node becomes runnable) and the index of the processor on which the node wants to be assigned. The latter is usually set to the index of the processor that its immediate ancestor is assigned. There are five public methods to manipulate these private variables.

`int availTime();`
`void setAvailTime(int` *t*);
`void setAvailTime();`
`void setPreferredProc(int` *i*);

The first three methods get and set the available time of the node. If no argument is given in `setAvailTime,` the available time is set to the earliest time when all ancestors are completed. The last two methods get and set the index of the processor on which the node is preferred to be scheduled.

15.10.2 Class HuGraph

Class HuGraph is the input APEG for Hu's level scheduler. It redefines three virtual methods of its parent classes.

`EGNode* newNode(DataFlowStar*` *s*, int *invoc*);

Creates a HuNode as a node in the APEG.

`void resetNodes();`

This method resets the variables of the HuNodes: visit flag, `waitNum,` the available time, and the index of the preferred processor.

`void sortedInsert(EGNodeList&` *nlist*, `ParNode*` *n*, `int` *flag*);

In the ParGraph class, this method sorts the nodes in order of decreasing `StaticLevel` of nodes. Now, we redefine it to sort the nodes in order of increasing available time first, and decreasing the static level next.

15.10.3 Class HuScheduler

Class HuScheduler, derived from the ParScheduler class, is parallel to the DLScheduler class in its definition.

`HuScheduler(MultiTarget*` *t*, `const char*` *log*);

The constructor has two arguments: one for the multiprocessor target and the other for the log file name.

The HuScheduler class has a pointer to the HuParProcs object that will provide the details of the Hu's level scheduling algorithm.

`HuParProcs* parSched;`

This is the protected member to point to the HuParProcs object. That object is created in the following method:

`void setUpProcs(int` *num*);

This method first calls `ParScheduler::setUpProcs` and next creates a HuParProcs object. The HuParProcs is deallocated in the destructor.

`int scheduleIt();`

The scheduling procedure is exactly same as that of the Dynamic Level Scheduler except that the actual scheduling routines are provided by a HuParProcs object rather than a DLParProcs object. Refer to the `scheduleIt` method of class DLScheduler . Also note that the runnable nodes in this scheduling algorithm are sorted by their available time first.

`StringList displaySchedule();`

Displays the scheduling result textually.

15.10.4 Class HuParProcs

Class HuParProcs is derived from class DLParProcs so that it has the same constructor. While many scheduling methods defined in the DLParProcs class are inherited, some virtual methods are redefined to realize different scheduling decisions. For example, it does not consider the communication overhead to determine the processor that can schedule a node earliest. And it does not schedule communication resources. Another big difference is that Hu's level scheduling algorithm has a notion of global time clock. No node can be scheduled ahead of the global time. At each scheduling step, the global time is the same as the available time of the node at the head of the list of runnable nodes.

`void fireNode(DLNode*` *n*);

This redefined protected method sets the available time and index of the preferred processor of the descendants, if they are runnable after node *n* is completed. This is done before putting them into the list of runnable nodes (`sortedInsert` method of the HuGraph class).

`void scheduleSmall(DLNode*` *n*);

When this method is called, the node *n* is one of the earliest runnable nodes. We examine a processor that could schedule the node at the same time as the available time of the node. If the node is at the wormhole boundary, we examine the first processor only. If the node should be assigned to the same processor on which any earlier invocations were already assigned, we examine that processor whether it can schedule the node at that time or not. If no processor is found, we increase the available time of the node to the earliest time when any processor can schedule it, and put the node back into the list of runnable nodes. If we find a processor to schedule the node at the available time of the node, we assign and fire the node, then update the variables of the HuGraph. Recall that no communication overhead is considered.

15.11 Declustering Scheduler

Declustering scheduler is the most elaborate scheduler developed by Sih . This algorithm only applies to the homogeneous multiprocessor targets and it does not support wormholes nor parallel stars. Since it takes into account the global information of the graph, it may overcome the weaknesses of list schedulers which consider only local information at each scheduling step. It turns out that this scheduling algorithm is very costly since it involves recursive executions of the list scheduler with various assignment, and choosing the best scheduling result. The scheduling routine was originally written in LISP for the Gabriel system and was translated into C++. Since the algorithm itself is very complicated, the reader of the code is highly encouraged to read Sih's paper on the scheduling algorithm.

Class DeclustScheduler is derived from class ParScheduler. It has a constructor with two arguments as the ParScheduler class. The subclass of ParProcessors used in DeclustScheduler is DCParProcs. The DeclustScheduler maintains two kinds of DCParProcs instances, one to save the best scheduling result so far, and the other is for retrying list scheduling whose results will be compared with the best result so far. These two DCParProcs are created in

`void setUpProcs(int *num*);`

They are deleted in the destructor.

```
StringList displaySchedule();
```

Displays the best scheduling result obtained so far.

The overall procedure of the declustering algorithm is:

(1) Make elementary clusters of nodes. To make elementary clusters, we examine the output arcs of all branch nodes (a branch node is a node with more than one output arc) and the input arcs of all merge nodes (a merge node is a node with more than one input arc). Those arcs are candidates to be cut to make clusters. An arc is cut if the introduced communication overhead can be compensated by exploiting parallelism.

(2) We make a hierarchy of clusters starting from elementary clusters up to one cluster which includes all nodes in the APEG. Clusters with the smallest work-load will be placed at the bottom level of the hierarchy.

The above two steps are performed in the following protected method:

```
int preSchedule();
```

Before making clusters, it first checks whether the APEG has wormholes or parallel stars. If it finds any, it returns `FALSE`.

(3) We decompose the cluster hierarchy. We examine the hierarchy from the top. We assign two processors to each branch (son cluster) of the top node at the next level. Then, we execute list scheduling and save the scheduling result. We choose a son cluster with a larger work-load. Then, we introduce another processor to schedule two branches of the son cluster. Execute a different list scheduling to compare the previously best scheduling result, then save the better result. Repeat this procedure until all processors are consumed. It is likely to stop traversing the cluster hierarchy since all processors are consumed. At this stage, we compare the loads of processors and try to balance the loads within a certain ratio by shifting some elementary clusters from the most heavily loaded processor onto a lightly loaded processor.

(4) In some cases, we can not achieve our load-balancing goals by shifting clusters. We try to breakdown some elementary clusters in heavily loaded processors to lightly loaded clusters. This is cluster "breakdown".

(5) In each step of (3) and (4), we execute list scheduling to compare the previously best scheduling result. This is the reason why the declustering algorithm is computationally expensive. Finally, we get the best scheduling result. Based on that scheduling result, we make a final version of the APEG including all communication nodes (in the `finalizeGalaxy`

method of DCParProcs class). Note that we do not call the list scheduling algorithm in the ParProcessors class after we find out the best scheduling result since we already executed that routine for that result.

Steps (3), (4), and (5) are performed in the following public method:

```
int scheduleIt();
```

Many details of the scheduling procedure are hidden with private methods. The remaining section will describe the classes used for Declustering scheduling one by one.

15.11.1 Class DCNode

Class DCNode, derived from class ParNode, is an APEG node for the declustering scheduler. It has the same constructors with the ParNode class. This class does not have any protected members.

```
int amIMerge();
int amIBranch();
```

These methods return `TRUE` if this node is a merge node or a branch node.

```
DCCluster* cluster;
DCCluster* elemDCCluster;
```

These pointers point to the highest-level cluster and the current elementary cluster owning this node.

```
void saveInfo();
int getBestStart();
int getBestFinish();
```

The first method saves the scheduling information of this node with the best scheduling result, which includes the processor assignment, the scheduled time, and the completion time. The next two methods return the scheduled time and the completion time of the current node.

`int getSamples(DCNode*` *`destN`*`);`

Returns the number of samples transferred from the current node to the destination node *`destN.`* If no sample is passed, it returns 0 with an error message.

`DCNode* adjacentNode(DCNodeList&` *`nlist`*, int *`direction`*);

DCNodeList is derived from class EGNodeList just to perform type casting. This method returns an adjacent node of the current node in the given node list. If *`direction`* is 1, look at the ancestors, if -1, look at the descendants.

```
StringList print();
```

Prints the master star name and the invocation number of the node.

There are three iterators defined for DCNode class: `DCNodeListIter, DCAncestorIter,` and `DCDescendantIter.` As names suggest, they return the DCNode in the list, in the ancestors of a node, and in the descendants of a node.

15.11.2 Classes DCArc and DCArcList

Class DCArc represents an candidate arc in the APEG to be cut in making elementary clusters. It has a constructor with five arguments.

`DCArc(DCNode* src, DCNode* sink, int first, int second,int third);`

The first two arguments indicate the source and destination nodes of the arc. The remaining three arguments define a triplet of information used to help find the arcs to be cut. We call these arcs "cut-arcs". We consider a pair of a branch node and a merge node and two paths between them to determine if a pair of cut-arcs to parallelize these two paths is beneficial. The *first* argument is the sum of execution times of nodes preceding this arc, starting from the branch node. The *second* argument is the communication overhead for this arc. The *third* argument is the sum of execution times of nodes following from this arc to the merge node.

The five arguments given to the constructor can be retrieved by the following methods:

```
DCNode* getSrc();
DCNode* getSink();
int getF();
int getS();
int getT();
```

They can be printed by

```
StringList print();
```

The sink and the source nodes can be reversed, and can be copied from an argument DCArc by the following methods:

```
void reverse();
int operator==(DCArc& arc);
```

There are other public methods as follow:

```
DCArcList* parentList();
```

A DCArc will be inserted to a list of DCArcs, call DCArcList. This method returns the pointer to the list structure.

```
int betweenSameStarInvoc();
```

Returns `TRUE` or `FALSE`, based on whether this arc is between invocations of the same star.

Class DCArcList is derived from class SequentialList to make a list of DCArcs. It has a constructor with no argument and a copy constructor. The destructor deletes all DCArcs in the list.

```
void insert(DCArc* arc);
void append(DCArc* arc);
```

These methods to put *arc* at the front and the back of the list, respectively.

```
DCArc* head();
```

Returns the DCArc at the front of the list.

```
int remove(DCArc* arc);
```

Removes *`arc`* from the list.

`int member(DCArc*` *`arc`*`);`
Returns `TRUE` if the given DCArc is a member of the list.

`int mySize();`
Returns the number of DCArcs in the list.

`StringList print();`
It prints a list of DCArcs in the list.

There is a iterator for DCArcList, called DCArcIter, which returns a DCArc.

15.11.3 Class DCGraph

Class DCGraph, derived from class ParGraph, is an input APEG for DeclustScheduler. It has no explicit constructor.

`EGNode* newNode(DataFlowStar*, int);`
Creates a DCNode as an APEG node for DCGraph.

`DCNodeList BranchNodes;`
`DCNodeList MergeNodes;`
These lists store the branch nodes and merge nodes.

`int initializeGraph();`
This protected method initializes the DCGraph. It sets up the lists of branch nodes and merge nodes (`BranchNodes, MergeNodes),` and the list of initially runnable nodes. We sort these lists by the static levels of the nodes: the branch nodes are sorted by smallest static level first while the merge nodes are sorted by largest static level first. In this method, we also initialize the DCNodes, which includes the detection of the merge nodes that are reachable from the node and the branch nodes reachable to the node.

The remaining methods are all public.
`const char* genDCClustName(int` *`type`*`);`
Generate a name for the cluster. If *`type`* = 0, we prefix with "ElemDCClust" to represent an elementary cluster. Otherwise, we prefix with "MacroDCClust".

`StringList display();`
Displays the APEG with the lists of initially runnable nodes, the branch nodes, and the merge nodes.

`DCNode* intersectNode(DCNode*` *`d1`*, `DCNode*` *`d2`*, `int` *`direction`*`);`
This method returns a merge node with the smallest static level, reachable from both *`d1`* and *`d2`* if *`direction`* = 1. If *`direction`* = 0, it returns a branch node with the smallest static level, that can reach both *`d1`* and *`d2`* nodes.

`DCArcList* traceArcPath(DCNode*` *`branch`*, `DCNode*` *`src`*, `DCNode*` *`dest`*, `int` *`direc-tion`*`);`

This method makes a list of candidate cut-arcs between *`branch`* and *`dest`* nodes, and returns the pointer to the list. The second argument, *`src,`* is an immediate descendant of the *`branch`* node on the path to the *`dest`* node. If *`direction`* = 1, we reverse all arcs and find cut-arcs from *`dest`* to *`branch`* nodes.

`void addCutArc(DCArc*` *`arc`*);
This method adds a DCArc to a list of cut-arcs in DCGraph.

`void formElemDCClusters(DCClusterList&` *`EClusts`*);
In this method, we remove all cut-arcs in the APEG and make each connected component an elementary cluster. The argument *`EClusts`* is the list of those elementary clusters. We connect these clusters at the end.

`void computeScore();`
In scheduling stage (3) of the DeclustScheduler, we may want to shift clusters from heavily loaded processors to lightly loaded processors. To prepare this step, we compute the score of top-level clusters in that scheduling phase. The score of a cluster is the number of samples passed to other processors minus the number of samples passed inside the same processor along the cut-arcs within that cluster. The score indicates the cost of shifting a cluster due to communication.

`void commProcs(DCCluster*` *`clust`*, int* *`procs`*);
This method finds processors that *`clust`* communicates with. We set the component of the second argument array to 1 if that processor communicates with the cluster.

`void copyInfo();`
Used for saving the scheduling information if the most recent scheduling result is better than the previous ones.

15.11.4 Class DCCluster

Class DCCluster represents a cluster of nodes in the declustering algorithm. There is no protected member in this class. It consists of two DCClusters, called component clusters, to make a hierarchy of clusters. An elementary cluster has `NULL` component clusters. It is constructed by a one-argument constructor.

`DCCluster(DCNodeList*` *`node-list`*);
Makes the cluster contain all nodes from the list.

To make a macro cluster, we use the following constructor:
`DCCluster(DCCluster*` *`clust1`*, `DCCluster*` *`clust2`*);
The argument clusters become the component clusters of this higher level cluster. The cluster-arcs are established from the cluster-arcs of two component clusters by calling the following method:

`void fixArcs(DCCluster*` *`clust1`*, `DCCluster*` *`clust2`*);
In this method, arcs put inside this cluster are removed from the arcs of two argument clusters.

In both constructors, we compute the sum of execution times of all nodes in the cluster, which can be obtained by

```
int getExecTime();
DCCluster* getComp1();
DCCluster* getComp2();
```

The last two methods above return two component clusters.

`void setName(const char*` *`name`*`);`
`const char* readName();`

The above methods set and get the name of the cluster.

`void addArc(DCCluster*` *`adj`*`, int` *`numSample`*`);`

This method adds a cluster-arc that is adjacent to the first argument cluster with sample rate *`numSample.`*

`void setDCCluster(DCCluster*` *`clust`*`);`

Sets the `cluster` pointer of the nodes in this cluster to the argument cluster.

`void assignP(int` *`procNum`*`);`
`int getProc();`

The first method assigns all nodes in the cluster to a processor. The second returns the processor that this cluster is assigned to.

`void switchWith(DCCluster*` *`clust`*`);`

Switches the processor assignment of this cluster with the argument cluster.

`DCCluster* pullWhich();`

Returns the cluster with the smaller execution time between two component clusters, and pull it out.

`DCCluster* findCombiner();`

This method returns the best cluster, in terms of cluster-arc communication cost, to be combined. We break ties by returning the cluster with smallest execution time.

```
void broken();
int getIntact();
```

These methods indicate whether the cluster or its subclusters were broken into its components in the scheduling stage (3) of DeclustScheduler. The first method indicates that it happens. The second method queries whether it happens or not.

`int getScore();`
`int setScore(int` *`score`*`);`
`void resetMember();`

These methods get and set the score of the cluster. Refer to the `computeScore` method of the DCGraph class to see what the score of a cluster is. The last method resets the score to 0.

`StringList print();`

Prints the name of this cluster and the names of component clusters.

`~DCCluster();`

The destructor deletes the nodes in the cluster and cluster-arcs if it is an elementary cluster.

15.11.5 Class DCClusterList

Class DCClusterList, derived from class DoubleLinkList, keeps a list of clusters. It has no protected members. It has a default constructor and a copy constructor.

`void insert(DCCluster*` *`clust`*`);`
`void append(DCCluster*` *`clust`*`);`
`void insertSorted(DCCluster*` *`clust`*`);`

These methods put *`clust`* at the head and the back of the list. The last method inserts the cluster in order of increasing execution time.

```
DCClusterLink* firstLink();
DCCluster* firstDCClust();
DCCluster* popHead();
```

The above methods return the DCClusterLink and DCCluster at the head of the list. The last method removes and returns the cluster from the list. Class DCClusterLink is derived from class DoubleLink as a container of DCCluster in the DCClusterList. It has a public method to access the cluster called

`DCCluster* getDCClustp();`
`DCClusterLink* createLink(DCCluster*` *`clust`*`);`
`void removeDCClusters();`

Create a DCClusterLink and removes all clusters in the list, respectively.

```
void resetList();
void resetScore();
void setDCClusters();
```

The first two methods reset the scores of all clusters to 0. The first method also declares that each cluster is not broken. The third method resets the cluster pointer of the nodes of the clusters in the list.

`int member(DCCluster*` *`clust`*`);`

Returns `TRUE` or `FALSE`, based on whether the argument cluster is in the list or not.

`void findDCClusts(DCNodeList&` *`nlist`*`);`

Add to the list clusters that own the nodes of the argument list. If the number of clusters is 1, we break the cluster into two component clusters and put them into the list.

`int listSize();`

Returns the number of clusters in the list.

`StringList print();`

Prints the list of clusters.

There is an iterator associated with the DCClusterList called DCClusterListIter that returns a DCCluster. It can return a DCClusterLink by the `nextLink` method.

15.11.6 Class DCClustArc and class DCClustArcList

Class DCClustArc represents a cluster-arc. It has a constructor with two arguments:

`DCClustArc(DCCluster*` *neighbor*, `int` *nsamples*);

The first argument is the pointer to the neighboring cluster while the second argument sets the sample rate of the connection.

`DCCluster* getNeighbor();`
`void changeNeighbor(DCCluster*` *clust*);

These methods return the neighbor cluster and change to it.

`void changeSamples(int` *newsamps*);
`void addSamples(int` *delta*);
`int getSamples();`

The above methods modify, increment, and return the sample rate of the current arc.

`StringList print();`

Prints the name of the neighbor cluster and the sample rate.

Class DCClustArcList is derived from class SequentialList to make a list of cluster-arcs. It has four public methods.

`DCClustArc* contain(DCCluster*` *clust*);

Returns the DCClustArc that is adjacent to the argument cluster. If no cluster-arc is found in the list, return 0.

`void changeArc(DCCluster*` *oldC*, DCCluster* *newC*);

This method changes the pointer of neighbor cluster, *oldC,* in all cluster-arcs in the list to *newC.*

`void removeArcs();`

Deletes all cluster-arcs in the list.

`StringList print();`

Prints the list of DCClustArcs.

There is an iterator associated with the DCClustArcList, called DCClustArcListIter, which returns a DCClustArc.

15.11.7 Class DCParProcs

Class DCParProcs is derived from class ParProcessors. It has the same constructor and destructor with the ParProcessors class.

There is one protected method:

`ParNode* createCommNode(int` *i*);

Creates a DCNode to represent a communication code. The argument indicates the type of the node.

The other methods are all public, and support the main scheduling procedure described in the DeclustScheduler class .

`int commAmount();`

Returns the communication overhead of the current schedule.

`void saveBestResult(DCGraph* `*`graph`*`);`

This method saves the current scheduling information of the nodes as the best scheduling result.

`void finalizeGalaxy(DCGraph* `*`graph`*`);`

After all scheduling is completed, we make a final version of the APEG including all communication loads based on the best scheduling result obtained.

`void categorizeLoads(int `*`procs`*`);`

This method categorizes each processor as either heavily or lightly loaded. It sets an integer array, *`nprocs,`* 1 for heavy and -1 for light processors. The initial threshold is 50 processors are heavily loaded if all processors are loaded beyond a 75 maximum load. We regard at most one idle processor as lightly loaded.

`int findSLP(DCNodeList* `*`nlist`*`);`

This method finds the progression of nodes (regular or communication) in the schedule which prevents the makespan from being any shorter. We call this set of nodes and *schedule limiting progression: SLP* (refer to Sih’s paper). The SLP can span several processors and can’t contain idle times. If there are several SLPs it will return just one of them.

Chapter 16. Base Code Generation Domain and Supporting Classes

Authors: *Soonhoi Ha*

Other Contributors: *Michael C. Williamson*

This chapter explains the base classes for code generation, which are found in the `$PTOLEMY/src/domains/cg/kernel` directory. Not all classes in that directory are covered in this document. We instead concentrate on how to generate and organize the code, and which methods to use. There is a basic code generation domain, designated CG, from which other code generation domains are derived. The CG domain can be used by itself for the purpose of studying issues in control constructs and scheduling, without needing to generate code in any particular programming language.

A segment of code is formed in an instance of class CodeStream. Each CGTarget will have a list of CodeStreams, and will assemble them to generate the final code. A CGStar uses instances of class CodeBlock to form a code segment, which can be added to a CodeStream of the CGTarget after some processing.

A set of macros are defined which a star programmer may use in order to refer to variables without being concerned about resource allocation. For example, we may refer to the portholes of a star without knowing what physical resources are allocated to them.

16.1 Class CodeStream

Class CodeStream is publicly derived from class StringList, and is used to make a sequential stream of code. In class CGTarget, a base target class for code generation, there are two CodeStreams: `myCode` and `procedures.`

```
CodeStream myCode;
CodeStream procedures;
```

These are protected members of class CGTarget. They are the default entries in `codeStringLists`, the list of code streams that CGTarget maintains.

```
CodeStreamList codeStringLists;
```

This is a protected member of class CGTarget. We can add a CodeStream to `codeStringLists` by using the following method of class CGTarget:

`void addStream(const char* *name*, CodeStream* *code*);`

This is a public method of class CGTarget. The first argument is the name of the CodeStream, and the second argument is a pointer to the CodeStream. This method should be called in the constructor of a target class. If a target attempts to add a CodeStream with an existing name, an error will be signaled.

`CodeStream* getStream(const char* name=NULL);`

This is a public method of class CGTarget. This method returns a pointer to the CodeStream with the given name. If no stream with the given name is found, this method returns `NULL`. If *name*`=NULL`, a pointer to `defaultStream` is returned. Class CGStar has a corresponding method to get the CodeStream with the given name from the CGStar's target.

The following method allows CGStars to construct a new CodeStream and add it to the CGTarget's list of CodeStreams. Some of the possible uses for this method are:

- a group of CGStars can build a procedure together
- a CGStar can add control flow constructs at the end of the `mainLoop` code

`CodeStream* newStream(const char* name);`

This is a public method of class CGTarget. There is a corresponding protected method of CGStar. This method adds a new CodeStream with the given name to the `codeStringLists` member of the CGTarget, and returns a pointer to the new CodeStream.

Now we will explain the public methods and members of class CodeStream.

`int put(const char* code, const char* name=NULL);`

This method puts the given segment of code at the end of the CodeStream. Optionally, the name of the code segment can be given. If *name*`=NULL,` we append the code unconditionally. Otherwise, we check to see if code with the same name has already ben added, by examining the `sharedNames` member of the CodeStream. If no code segment with the same name is found, the code segment is appended. This method returns `TRUE` if code was successfully added to the stream, `FALSE` otherwise.

`UniqueStringList sharedNames;`

This is a public member of class CodeStream. It is used to store the names of code segments added by name to the CodeStream. Class UniqueStringList is privately derived from class StringList.

`void initialize();`

This is a public method of class CodeStream. It is used to initialize both the code list and `sharedNames.`

`int isUnique(const char* name);`

This is a public method of class UniqueStringList. This method returns `FALSE` if the argument string already exists in the UniqueStringList. If not, then the method adds the string to the list and returns `TRUE`.

Class CodeStreamList contains a list of CodeStreams. It is publicly derived from class NamedList since each CodeStream is assigned a name. There are four public methods in the CodeStreamList class:

```
int append(CodeStream* stream, const char* name);
int add(const char* name, CodeStream* stream);
CodeStream* get(const char* name) const;
int remove(const char* name);
```

The first two methods append a CodeStream to the list. They differ from each other in the order of arguments. The third method returns a CodeStream with the given name while the last method removes the CodeStream with the given name from the list.

16.1.1 Class NamedList

Class NamedList is privately derived from class SequentialList, and is used to make a list of objects with names. It has a default constructor. The destructor deletes all objects in the list. There are no protected members in this class.

```
int append(Pointer object, const char* name);
void prepend(Pointer object, const char* name);
```

These methods put an object, *object*, with name *name* at the end and the beginning of the list, respectively. In the first method, we may not append multiple objects with the same name. If an object with the same name exists in the list, `FALSE` is returned. On the other hand, the second method allows multiple objects with the same name to be prepended. Only the most recently prepended object will be visible.

```
Pointer get(const char* name=NULL);
```

This method returns the object with the given name. If no name is given, it returns the object at the head of the list. If no object is found, it returns `NULL`.

```
int remove(const char* name=NULL);
```

This method removes the object with the given name. If no name is given, it removes the first object at the head of the list. If no object is found it returns `FALSE`, otherwise it returns `TRUE`.

There is an iterator class associated with the NamedList class, called NamedListIter. It returns a pointer to the next object in the list as it iterates through the list.

16.2 Class CodeBlock and Macros

Class CodeBlock stores a pointer to text in its constructor.

```
CodeBlock(const char* text);
```

It is up to the programmer to make sure that the argument text lives as long as the code-block is used.

There are four public methods defined to access the text:

```
void setText(char* line);
const char* getText();
operator const char*();
void printCode();
```

The first method sets the text pointer in the CodeBlock. The next two methods return the text this CodeBlock points to. The last method prints the code to the standard output.

A star programmer uses the `codeblock` directive in the preprocessor language file to define a block of text. In a CodeBlock, the programmer uses the following macros in order to refer to the star ports and variables without needing to be concerned about resource management or name conflicts:

`$val(`*`name`*`)`
  Value of a state

`$size(`*`name`*`)`
  Buffer size of a state or a porthole

`$ref(`*`name`*`)`
  Reference to a state or a porthole

`$ref(`*`name, offset`*`)`
  Reference with offset

`$label(`*`name`*`)`
  Unique label inside a codeblock

`$codeblockSymbol(`*`name`*`)`
  Another name for $label

`$starSymbol(`*`name`*`)`
  Unique label inside a star

`$sharedSymbol(`*`list`*`, `*`name`*`)`
  Unique label for set list, name pair

These macros are resolved into code after resources are allocated or unique symbols are generated.

A CodeBlock defined in a CGStar is put into a CodeStream of the target by the following methods of the CGStar class:

```
int addCode(const char* code, const char* stream=NULL,
            const char* name=NULL);
int addProcedure(const char* code, const char* name);
```

These are protected methods of class CGStar. The first method puts a segment of code, *`code,`* at the end of the target's CodeStream with name *`stream`*. If the name of the CodeStream is not given, the method uses the `myCode` stream of the target. The second method uses the `procedure` CodeStream of the target. The argument *`name`* of both methods is optionally used to specify the name of the code. If the code is successfully added, the methods return `TRUE`, otherwise they return `FALSE`. Before putting the code at the end of the CodeStream, the code is processed to resolve any macros by the application of the `processCode` method:

```
StringList processCode(CodeBlock& cb);
StringList processCode(const char* code);
```

These methods are both protected and essentially equivalent since the first method calls the second method. They scan the code, word by word, and copy it into a StringList. If a macro is found, the macro is expanded through a call to `expandMacro` before being copied to the StringList. Testing can be done to check whether a word is a macro or not by comparing the first character with the result of the following method:

```
virtual char substChar() const;
```

This method is a virtual protected method of class CGStar. It is used to return the special character that marks the beginning of a macro in a code block. In the CGStar class, it returns the dollar sign character, `$`.

```
virtual StringList expandMacro(const char* func,
                               const StringList& argList);
```

This is a virtual protected method of class CGStar. It is used to expand a macro with the given name *`func.`* The argument list must be passed by reference so that the StringList will not be consolidated. It is virtual so that derived classes can define more macros. A macro is identified by the following method:

```
int matchMacro(const char* func, const StringList& argList,
               const char* name, int argc);
```

This protected method of class CGStar returns `TRUE` if the first argument *`func`* matches with the third argument *`name`*, and the number of arguments in *`argList`* is the same as the count *`argc`*.

Based on the particular macro being applied, one of the following protected methods may be used to expand the macro:

```
virtual StringList expandVal(const char* name);
StringList expandSize(const char* name);
virtual StringList expandRef(const char* name);
virtual StringList expandRef(const char* name, const char* offset);
```

The first three methods expand the $val, $size, and $ref macros. The fourth method expands the $ref macro when it has two arguments. These virtual methods should be redefined in derived classes. In particular, the last two methods must be redefined in derived classes because in class CGStar they generate error messages. The other macros deal with unique symbols within the scope of a code block, within a star, and within a set of symbols. More will be said about these in the next subsection .

When an error is encountered while expanding macros or processing code blocks, the following methods should be called to generate an error message:

```
void macroError(const char* func, const StringList& argList);
void codeblockError(const char* p1,  const char* p2="");
```

The arguments of the second method provide the text of the error message.

16.3 Class SymbolList and Unique Symbol Generation

In order to generate a unique symbol within a scope, a list of symbols should be made for that scope. For example, the CGStar class has two protected members which are SymbolLists, `starSymbol` and `codeblockSymbol.`

```
SymbolList starSymbol;
SymbolList codeblockSymbol;
```

Class SymbolList is derived from class BaseSymbolList. Class BaseSymbolList is privately derived from class NamedList. A BaseSymbolList keeps two private members which are used to create a unique name for each symbol in the list: a separator and a counter:

`BaseSymbolList(char` *`sep`*`='_', int*` *`count`*`=NULL);`

The first argument of the constructor is used to set the separator, and the second argument is used to set the pointer of the count variable. These two variables can be set independently by invoking the following methods:

`void setSeparator(char` *`sep`*`);`
`void setCounter(int*` *`count`*`);`

When we append or insert a new symbol into the list, we create a unique name for that symbol by appending a separator followed by the counter value to the argument symbol, and then return the unique name:

`const char* append(const char*` *`name`*`);`
`const char* prepend(const char&` *`name`*`);`
`const char* get(const char*` *`name`*`=NULL);`

This last method returns the unique symbol with the given name. If no name is given, it returns the first symbol in the list.

`int remove(const char*` *`name`*`=NULL);`

This method removes the unique symbol with the given name. If no name is given, it removes the first symbol in the list. It returns `FALSE` if no symbol is removed.

Symbols in the list are deleted in the destructor, and in the following method:
`void initialize();`

The public method
`Stringlist symbol(const char*` *`string`*`)`
makes a unique symbol from the supplied argument by adding the separator and a unique counter value to the argument string.

Class SymbolList is privately derived from class BaseSymbolList with the same constructor and a default destructor. Class SymbolList uncovers only three methods of the base class:

```
BaseSymbolList::setSeparator;
BaseSymbolList::setCounter;
BaseSymbolList::initialize;
```

Class SymbolList adds one additional method:
`const char* lookup(const char*` *`name`*`);`

If a unique symbol with the given name exists, this method returns that unique symbol. Otherwise, it creates a unique symbol with that name and puts it into the list.

Recall that the CGStar class has two SymbolLists. The macros `$codeblockSymbol`, `$label`, and `$starSymbol` are resolved by the `lookUp` method of the `codeblockSymbol` and `starSymbol` SymbolLists, based on the scope of the symbol. If the symbol already exists in the SymbolList, it returns that unique symbol. Otherwise, it creates a unique symbol in the scope of interest.

If we want to generate a unique symbol within the file scope, we use a scoped symbol list defined in the target class.

```
ScopedSymbolList sharedSymbol;
```

It is a protected member of the CGTarget class. Class ScopedSymbolList is privately derived from class NamedList to store a list of SymbolLists. It has the same constructor as the base class.

```
void setSeparator(char set);
void setCounter(int* count);
```

These methods in class ScopedSymbolList are used to set the separator and the counter pointer of all SymbolLists in the list.

```
const char* lookup(const char* scope, const char* name);
```

In this method, the first argument determines the SymbolList in the list named *scope,* and the second argument determines the unique symbol within that SymbolList. If no SymbolList is found with the given name, we create a new SymbolList and insert it into the list.

The SymbolLists in the list are deleted in the destructor and in the following method:

```
void initialize();
```

Now we can explain how to expand the last macro defined in the CGStar class: $sharedSymbol. The first argument of the macro determines the StringList and the second argument accesses the unique string in that StringList. It is done by calling the following protected method in the CGStar class:

```
const char* lookupSharedSymbol(const char* scope, const char* name);
```

This method calls the corresponding method defined the CGTarget class.

The CGTarget class has another symbol list:

```
SymbolStack targetNestedSymbol;
```

It is a protected member used in generating unique nested symbols. Class SymbolStack is privately derived from class BaseSymbolList. It has the same constructor as the base class and has a default destructor.

For stack operation, class SymbolStack defines the following two methods:

```
const char* push(const char* tag="L");
StringList pop();
```

These methods push the symbol with given name onto the top of the list and pop the symbol at the top of the list off of the list, respectively.

This class also exposes several methods of the base class:

```
BaseSymbolList::get;
BaseSymbolList::setSeparator;
BaseSymbolList::setCounter;
BaseSymbolList::initialize;
BaseSymbolList::symbol;
```

In this section, we have explained various symbol lists. The separator and the counter are usually defined in the CGTarget class:

```
char separator;
int counter;
```

The first is a public member in class CGTarget, and is set in the constructor. The second is a private member in class CGTarget, and is initialized to zero in the constructor. The counter value is accessed through the following public method:

```
int* symbolCounter();
```

16.4 Class CGGeodesic and Resource Management

When we generate assembly code, we have to allocate memory locations to implement the portholes and states of each star. For high-level language generation, we assign unique identifiers to them. It is rather easy to allocate resources for states since state requirements are visible from the star definition: type, size and name. In this section, we will focus on how to determine the buffer size for the porthole connections.

We allocate a buffer for each connection. We do not assume in the base class, however, that the buffer is owned by the source or by the destination porthole. Instead, we use methods of the CGGeodesic class. Before determining the buffer sizes, we obtain information about how many samples are accumulated on each CGGeodesic by simulating the schedule. This is for the case of synchronous dataflow (SDF) semantics with static scheduling.

The minimum buffer requirement of a connection may be determined by considering only local information about the connection:

```
int minNeeded() const;
```

This method returns that minimum buffer size. It is a protected member of class CGGeodesic.

We do not want to allocate buffers for connections when it is unnecessary. For example, the output portholes of a Fork star can share the same resource with the Fork star's input porthole. A Gain star with unity gain is another trivial example. Therefore, we pay special attention to stars of type Fork. Without confusion, we refer to a star as a *Fork* star if its outputs can share the same resource with its input. In the CGStar class, we provide the following methods:

```
int isItFork();
void isaFork();
virtual void forkInit(CGPortHole& input, MultiCGPortHole& output);
virtual void forkInit(CGPortHole& input, CGPortHole& output);
```

The first is a public method of class CGStar. The rest are protected methods of class CGStar. The first method queries whether the star is a Fork star. The second method is used to declare that the star is a Fork star. If it is, we can call either one of the last two methods, based on whether the output is a MultiPortHole or not. In those methods, we shift delays from a Fork's input port to the output ports, and set the `forkSrc` pointer of the output ports to point to the Fork's input port. The Fork's input port keeps a list of the output ports in its `forkDests` member. We apply this procedure recursively in the case of cascaded Forks.

```
CGPortHole* forkSrc;
SequentialList forkDests;
```

These are protected members of class CGPortHole. The first one is set by the following public method:

`void setForkSource(CGPortHole* p, int cgPortHoleFlag=TRUE);`

The first argument is the input porthole of the Fork star and the port this is being called on should be an output porthole when we call this method.

`int fork() const;`

This is a public method of class CGPortHole which returns `TRUE` if it is an input porthole of a Fork star.

Class CGGeodesic provides two methods to return the Fork input port if it is at a Fork output port. Otherwise these methods return `NULL`.

```
CGPortHole* src();
const CGPortHole* src() const;
```

These two methods are protected and differ from each other in their return type.

Now we will explain more of the methods of class CGGeodesic.

`int forkType() const;`

This public method of class CGGeodesic indicates the type of the current CGGeodesic. If it is at a Fork input, it is `F_SRC.` If it is at a Fork output, it is `F_DEST.`

`int forkDelay() const;`

This public method of class CGGeodesic returns the amount of delay from the current Geodesic up to the fork buffer that this Geodesic refers to. If it is not associated with a fork buffer, it returns 0.

We do not allocate a buffer to a CGGeodesic if it is `F_DEST.`

```
int localBufSize() const;
int bufSize() const;
```

The above public methods of class CGGeodesic return the buffer size associated with this CGGeodesic. While the first method returns 0 if the CGGeodesic is at a Fork output, the second method returns the size of the fork buffer. The actual computation of the buffer size is done by applying the following method:

`virtual int internalBufSize() const;`

This protected method of class CGGeodesic returns 0 with an error message if the schedule has not yet been run. If this CGGeodesic is a `F_SRC,` the minimum size is set to the maximum buffer requirements over all fork destinations. If there are delays or if old values are used, we may want to use a larger size so that compile-time indexing is supportable. The buffer size must divide the total number of tokens produced in one execution. To avoid modulo addressing, we prefer to use the *LCM* value of the number of samples consumed and produced during one iteration of the schedule. Since this may be wasteful, we check the extra buffer size required for linear addressing with the `wasteFactor.` If the waste ratio is larger then `wasteFactor,` we give up on linear addressing.

`virtual double wasteFactor() const;`

In the CGGeodesic class, this method returns 2.0. If a derived class wants to enforce linear addressing as much as possible, it should set the return value to be large. To force the minimum buffer memory size to be used, the return value should be set to 1.0.

`void initialize();`

This public method of class CGGeodesic initializes the CGGeodesic.

Refer to class CGPortHole for more information on resource management.

16.5 Utility Functions

There are several utility functions defined in the CG domain for aiding in code generation. Here we describe just a few of them:

```
char* makeLower(const char* name);
int rshSystem(const char* hostname, const char* command,
              const char* directory=NULL);
```

The above functions are defined in the file CGUtilities.h. The first method returns a dynamically allocated string that is a lower-case version of the argument string. The second method is used to execute a remote shell command, *`command,`* in the *`directory`* on the machine *`hostname.`* We use the `xon` command instead of `rsh` in order to preserve any X-Window environment variables.

16.6 Class CGStar

In this section, we will explain additional class CGStar members and methods not described above in this chapter. Class CGStar has a constructor with no arguments. Class CGStar is derived from class DynDFStar, and not from class SDFStar, so that BDF and DDF code generation may be supported in the future.

There is an iterator to enumerate the CGPortHoles of a CGStar: class CGStarPortIter. The `next()` and `operator++` methods return type `CGPortHole*`.

16.6.1 CGStar Protected Methods and Members

Protected members related to CodeStream, SymbolList, and resource management can be found in earlier sections of this chapter.

`virtual void outputComment(const char* msg, const char* stream=NULL);`

This method adds a comment *`msg`* to the target *`stream.`* If no target stream is specified, the `myCode` stream is used.

`StringList expandPortName(const char* name);`

If the argument specifies the name of a MultiPortHole, the index may be indicated by a State. In this case, this method gets the value of the State as the index to the MultiPortHole and returns a valid MultiPortHole name. This method is used in the `expandSize` method.

`void advance();`

This method updates the offset variable of all PortHoles of the CGStar by the number of samples consumed or produced. It calls the `advance` method of each PortHole.

`IntState procId;`

This is an integer state to indicate processor assignment for parallel code generation. By default, the value is -1 to indicate that the star is not yet assigned.

`int dataParallel`

This is a flag to be set if this star is a wormhole or a parallel star.

`Profile* profile;`

This is a pointer to a Profile object, which can be used to indicate the local schedule of a data parallel star or macro actor. If it is not a parallel star, this pointer is set `NULL`.

`int deferrable();`

When constructing a schedule for a single processor, we can defer the firing of a star as long as possible in order to reduce the buffer requirements on every output arc. In this method, we never defer a Fork star, and always defer any non-Fork star that feeds into a Fork. This prevents the resulting fork buffer from being larger than necessary, because new tokens are not added until they must be.

16.6.2 CGStar Public Methods

`const char* domain() const;`
`int isA(const char* `*`class`*`);`

The first method returns `"CG"`. The second method returns `TRUE` if the argument *`class`* is `CGStar` or a base class of CGStar.

`int isSDF() const;`

Returns `TRUE` if it is a star with SDF semantics (default). For BDF and DDF stars, it will return `FALSE`.

`virtual void initCode();`

This method allows a star to generate code outside the main loop. This method will be called after the schedule is created and before the schedule is executed. In contrast, the `go()` method is called during the execution of the schedule, to form code blocks into a main loop body.

`int run();`

In CG domains, this method does not perform any actual data movement, but executes the `go()` method followed by the `advance()` method.

`CGTarget* cgTarget();`
`int setTarget(Target* `*`t`*`);`

These methods get and set the pointer to the target to which this star is assigned. When we set the target pointer, we also initialize the SymbolLists and the CodeStream pointers. If this method is successful, it returns `TRUE`, otherwise it returns `FALSE`.

`virtual int isParallel() const;`
`virtual Profile* getProfile(int `*`ix`*`=0);`

The first method returns `TRUE` if this star is a wormhole or a parallel star. If it is parallel, the second method returns the pointer to a Profile, indexed by the argument. A parallel star stores its internal scheduling results in a Profile object .

```
int maxComm();
```

Returns the maximum communication overhead with all ancestors. It calls the `commTime` method of the target class to obtain the communication cost.

```
virtual void setProcId(int i);
virtual int getProcId();
```

These methods set and get the processor ID to which this star is assigned.

16.7 Class CGPortHole

Class CGPortHole is derived from class DynDFPortHole in order to support non-SDF dataflow stars as well as SDF stars. Methods related to Fork stars are described in a the section above on Resource Management .

In this section, we will categorize the members and methods of CGPortHole into four categories: buffer management, buffer embedding, geodesic switching, and others.

16.7.1 Buffer Management

A CGPortHole is connected to a buffer after resource allocation. A CGPortHole maintains an offset index into the buffer in order to identify the current position in the buffer where the porthole will put or get the next sample:

```
int offset;
```

This is a protected member used for indexing into the buffer connected to this port.

The methods described in this subsection are all public:

```
unsigned bufPos() const;
```

Returns `offset`, the offset position in the buffer.

```
virtual int bufSize() const;
virtual int localBufSize() const;
```

Both methods returns the size of buffer connected to this porthole. In the CGPortHole base class, they call the corresponding methods of class CGGeodesic. Recall that the second method returns 0 when it is a Fork output. If a porthole is at the wormhole boundary, both return the product of the sample rate and the repetition count of the parent star.

```
virtual void advance();
```

This method is called by `CGStar::advance()`. After the parent star is executed, we advance the offset by the number of samples produced or consumed. The offset is calculated modulo the buffer size, so that it is wrapped around if it reaches the boundary of the buffer.

16.7.2 Buffer Embedding

As a motivating example, let’s consider a DownSample star. If we allocate separate buffers to the input and output ports, the buffer size of the input port will be larger than that of the output port. Also, we will need to perform an unnecessary copy of samples. We can improve this situation by allocating one buffer at the input site and by indicating that a subset of that buffer is the image of the output buffer. We call this relationship *embedding*: the larger input buffer is *embedding* the smaller output buffer, and the smaller output buffer is *embedded* in the larger input buffer. Unlike the Fork buffer, the sizes of input and output embedded buffers are different from each other. Therefore, we must specify at which position the embedded buffer begins in the larger embedding buffer. We use this embedding relationship to implement Spread and Collect stars in the CGC domain, without increasing the buffer requirements. For example, the output buffers of a Spread star are embedded in the input buffer of the star, starting from different offsets.

```
CGPortHole* embeddedPort;
int embeddingFlag;
```

These are protected members to specify embedding relationships. The first one points to the embedding port which this PortHole is embedded in. The second member indicates the starting offset of embedding. The last member indicates whether this porthole is an embedding port or not.

The following are public methods related to embedding.

```
CGPortHole* embedded();
int whereEmbedded();
int embedding();
```

These methods return the protected members described above, respectively.

`void embed(CGPortHole& p, int i=-1);`

This method establishes an embedding relationship between this port and the argument port *p*. This porthole becomes an embedding porthole, and the argument porthole becomes an embedded porthole. The second argument specifies the starting offset.

`void embedHere(int offset);`

This method, when called on an embedded porthole, changes the starting offset its embedded buffer in the embedding buffer.

16.7.3 Geodesic Switching

In the specification block diagram, a PortHole is connected to a Geodesic. In code generation domains, we usually allocate one resource to the Geodesic so that the Geodesic's source and destination ports can share the same resource (Note that this is not a strict requirement). After resource allocation, we may want to alias a porthole to another porthole, and therefore associate it with a resource other than the allocated resource. To do that, we switch the pointer of the Geodesic to another Geodesic.

```
virtual void switchGeo(Geodesic* g);
virtual void revertGeo(Geodesic* g);
```

Both methods set the Geodesic pointer to the argument *g*. There is a flag to indicate whether this port has switched its Geodesic or not. The first method sets the flag to `TRUE` while the second method resets the flag to `FALSE`. Both methods are virtual since in derived classes we may need to redefine the behavior, perhaps by saving the original Geodesic, which is not the default behavior. The flag is queried by:

```
int switched() const;
```

If the Geodesic is switched in this port, we have to reset the geodesic pointer of this port to `NULL` in the destructor in order to prevent attempts to delete the same Geodesic multiple times. Also, we have to make sure that both ends of a Geodesic do not switch their Geodesic, in order to prevent orphaning the geodesic and causing a memory leak.

16.7.4 Other CGPortHole Members

Class CGPortHole has a constructor with no argument which resets the member variables. In the destructor, we clear the `forkDests` list and remove the pointer to this porthole from the `forkDests` list of the `forkSrc` port. All members described in the subsection are public.

```
CGGeodesic& cgGeo() const;
```

This method returns a reference to the Geodesic after type casting.

```
void forceSendData();
void forceGrabData();
```

These methods put and get samples to and from the Geodesic at the wormhole boundary. They are used when the inside code generation domain communicates by the wormhole mechanism.

16.7.5 CGPortHole Derived Classes

Class InCGPort and class OutCGPort are publicly derived from class CGPortHole. They are used to indicate by class type whether a porthole is an input port or an output port.

Class MultiCGPort is derived from class MultiDFPort. It has a protected member `forkSrc` to point to the Fork input if its parent star is a Fork star. It has a default destructor.

```
CGPortHole* forkSrc;
```

There are two public methods related to this protected member:

```
void setForkBuf(CGPortHole& p);
void forkProcessing(CGPortHole& p);
```

The first method sets `forkSrc` to point to the argument port. The second method sets the `forkSrc` pointer of the argument port to point to the `forkSrc` of this MultiCGPort.

Two classes are publicly derived from MultiCGPort: MultiInCGPort and MultiOutCGPort. They both have the following public method:

```
PortHole& newPort();
```

This method creates an InCGPort or an OutCGPort depending on whether it is an input or an output MultiCGPort.

Chapter 17. Target

Authors: *Soonhoi Ha*

Other Contributors: *John S. Davis II*

Target has a clear meaning in code generation domains, a model of the target machine for which code will be generated. Class CGTarget is the base class for all code generation targets whether it is a single processor target or a multiprocessor target. Class MultiTarget, derived from class CGTarget, serves as the base target for all multiprocessor targets. For single processor targets, we have AsmTarget and HLLTarget to distinguish assembly code generation targets and high level language generation targets. If we generate assembly code for a target, the target will be derived from class AsmTarget. If we generate a high level language code, the target will be derived from HLLTarget. For detailed discussion for Target hierarchy, refer to [4] in References .

In this chapter, we will describe class CGTarget and some base multiprocessor targets since we focus on multiprocessor code generation. Refer to other sources for AsmTarget and other high level language targets.

17.1 Class CGTarget

Class CGTarget is derived from class Target. It has a four-argument constructor.

```
CGTarget(const char* name, const char* starclass,
      const char* desc, char sep);
```

The first argument is the name of the target and the second argument is the star class that this target can support. The third argument is the description of this target. The last one is a separator character for unique symbol generation.

There are two protected states in the CGTarget:

```
StringState destDirectory;
IntState loopingLevel;
```

The first state indicates where to put the code file. The second state determines which scheduler is used in case this target is a single processor target. By default, *loopingLevel* = 0 and we do not try looping. If *loopingLevel* = 1, we select Joe’s loop scheduler. Otherwise, we use the most complicated loop scheduler.

At the top level, three methods of the Target class are called in sequence: `setup, run,` and `wrapup.`

```
void setup();
```

In this method, we do the following tasks:

(1) Initialize `myCode` and `procedure` code stream.

(2) Select a scheduler if no scheduler is selected yet.

At this stage, we check whether the galaxy is assigned or not. In multiprocessor targets, a child target is not assigned a galaxy until a sub-univers is created. If the galaxy is not assigned, return.

(3) Reset the symbol lists.

(4) If we are the top-level target, initialize the galaxy (which performs preinitialization and initialization of the member blocks, including HOF star expansion and porthole type resolution). Then modify the galaxy if necessary by calling *`modifyGalaxy`*. The base class implementation of modifyGalaxy splices in type conversion stars where needed, if the domain has supplied a table of type conversion stars. (If there is no table, type conversion is presumed not needed. If there is a table, and a type mismatch is found that the table has no entry to fix, then an error is reported.) Some derived domains redefine *`modifyGalaxy`* to perform additional transformations. For example, in AsmTarget we insert some stars (CircToLin, LinToCirc) at loop boundaries to change the type of buffer addressing in case a loop scheduling is performed.

`virtual int modifyGalaxy();`

Is a protected method.

(5) If it is a child target, the schedule was already made at this stage from a parallel scheduler of the parent multiprocessor target. Otherwise, we initialize and schedule the graph.

(6) If it is a child target or it is not inside a wormhole, return. Otherwise, we first adjust the sample rate of the wormhole portholes (`adjustSampleRates).` Then, we generate and download code: `generateCode` and `wormLoadCode.`

`void adjustSampleRates();`

This method is a protected method to be called when this target is inside a wormhole. After scheduling is performed, we need to multiply the sample rate of wormhole portholes by the repetition count of the stars inside the wormhole connected to the porthole.

`virtual void generateCode();`

This method guides the overall procedure to generate code for single processor targets. The procedure is as follows:

(1) If this target is a child target, call `setup` to initialize the variables. Copy the symbol counter `(symbolCounter)` of the parent target to the symbol counter of this target to achieve a unique symbol in the system scope.

(2) We compute buffer sizes, allocate memory, etc: `allocateMemory.`

`virtual int allocateMemory();`

This method is protected. It does nothing and returns TRUE in this base class.

(3) Call the method `generateCodeStreams().` This method will be described later.

(4) Organize the CodeStreams into a single code stream and save the result to the `myCode`

stream: `frameCode.`

`virtual void frameCode();`

This method is a protected method. It does nothing in this base class.

(5) If this target is not a child target, write the generated code to a file: `writeCode.`

`virtual void writeCode(const char*` *`name`* = NULL);

This is a public method to write the `myCode` stream to the argument file. If no argument is given, use "code.output" as the default file name.

(6) If it is a child target, copy the symbol counter to that of the parent target.

The methods described above for code generation are all virtual methods. They will be redefined in the derived targets.

The method

`void CGTarget::generateCodeStreams();`

does the following things:

(1) Write initial code.

`virtual void headerCode();`

In this base class, this protected method writes the header comment to the `myCode` Code-Stream.

```
virtual StringList headerComment(const char* begin = NULL,
      const char* end = "", const char* cont = NULL);
```

This method is a public virtual method to generate the header comment in the code. In this base class, the head comments include the user id, code creation date, target name, and the galaxy name. The arguments are passed to the `comment` method.

```
virtual StringList comment(const char* cmt, const char* begin =
NULL, const char* end = "", const char* cont = NULL);
```

This public method generates a comment from a specified string *`cmt.`* We prepend *`begin`* and append *`end`* to the string. If *`begin`* is NULL, we prepend '#' as a shell-stype comment. If *`cont`* is specified, multi-line comments are supported.

(2) We do initialization for code generation: for example, compute offsets of portholes and call `initCode` methods of stars: `codeGenInit`.

`virtual int codeGenInit();`

is a protected method. It does nothing and returns TRUE in this base class.

(3) Generate the code for the main loop: `mainLoopCode.`

`virtual void mainLoopCode();`

In this method we first compute the number of iterations. If this target is inside a wormhole, the number is -1 indicating an infinite loop. Otherwise, the `stopTime` of the scheduler determines the number of iterations. In this base class, we call the following five methods sequentially: `beginIteration,` `wormInputCode` if inside a wormhole, `com-`

`pileRun`, `wormOutputCode` if inside a wormhole, and `endIteration`. In the derived class, this sequence may be changed .

```
void beginIteration(int numiter, int depth);
void endIteration(int numiter, int depth);
```

These public methods form the head or ending of the main loop. The arguments of both methods are the number of iteration and the depth of the loop. In the main loop, the depth is set 0.

```
virtual void wormInputCode();
virtual void wormOutputCode();
virtual void wormInputCode(PortHole& p);
virtual void wormOutputCode(PortHole& p);
```

The above methods are all public. They generate code at the wormhole boundary if the target resides in a wormhole. The last two methods generate code for the argument porthole that is at the wormhole boundary. In this base class, put comments in `myCode` CodeStream indicating that the methods are successfully executed. They should be redefined in the derived classes to be useful. The first two methods traverse all portholes at the wormhole boundary to use the last two methods.

```
virtual void compileRun(SDFScheduler* sched);
```

This protected method calls `compileRun` of the argument scheduler. By default, this method calls `go` methods of all stars in the scheduled order to generate code in `myCode` CodeStream.

(4) Call `wrapup` methods of stars to generate code after the main loop, but still inside the main function.

(5) Add more code if necessary: `trailerCode`

```
virtual void trailerCode();
```

This protected method does nothing in this base class.

The method

```
virtual int wormLoadCode();
```

is protected. It downloads code to the target machine and starts executing it if the target resides in a wormhole. In this base class, we just display the code.

Now, we discuss the `run` method.

```
int run();
```

If the target is not inside a wormhole, it generates code by calling `generateCode` as explained above. Otherwise, we do the transfer of data to and from the target since this method will be called when the wormhole is executed: `sendWormData` and `receiveWormData` in sequence.

```
virtual int sendWormData();
virtual int receiveWormData();
virtual int sendWormData(PortHole& p);
virtual int receiveWormData(PortHole& p);
```

The above methods are all protected. They send and receive samples to this target when run inside a wormhole. The argument is the porthole of the interior star at the wormhole boundary. If no argument is given, send and receive for all the appropriate portholes. In this base class, we generate comments to indicate that these methods are successfully called.

`void wrapup();`

In derived classes, wrapup will generate code to finalize, download, and run the code. This CGTarget class just displays the code.

So far, we have explained the three top level methods of the CGTarget class. Methods related to the CodeStream and unique symbol generations can be found in the previous chapter. We will describe the remaining members.

17.1.1 Other CGTarget protected members

`char* schedFileName;`

The name of the log file in case a loop scheduling is taken. By default, the name is set to "`schedule.log`".

`int noSchedule;`

This is a flag to be set to TRUE if scheduling is not needed in the setup stage. This flag will be set when the schedule is copied from `copySchedule` method in parallel code generation. By default, this flag is set FALSE.

`StringList indent(int` *`depth`*`);`

This method returns a list of spaces for indenting. The number of spaces is 4 per each *`depth.`*

`void switchCodeStream(Block*` *`b`*`, CodeStream*` *`s`*`);`

This method is set to the current `myCode` pointer of the argument block *`b`* to *`s`* CodeStream. If *`b`* is a galaxy, perform this for all component stars.

17.1.2 Other CGTarget public members

`static int haltRequested();`

Returns TRUE if error is signaled while Ptolemy is running.

`int inWormhole();`
`int isA(const char*` *`class`*`);`

Is a standard `isA` method for type identification.

Returns TRUE or FALSE, based on whether the target is inside a wormhole or not.

`Block* makeNew() const;`

Create a new, identical CGTarget. Internal variables are not copied.

`virtual int incrementalAdd(CGStar*` *`s`*`, int` *`flag`* `= 1);`

This method is called when we add code for the argument star *`s`* incrementally. If *`flag`* is 1 (default), we allocate memory for the star, and call `setup`, `initCode`, `go`, and `wra-`

`pup` of the star. If *`flag`* is 0, we just call `go` method of that star. In this base class, generate an error message.

`virtual int insertGalaxyCode(galaxy*` *`g`*, `SDFScheduler*` *`sched`*);
This method inserts the code for the argument galaxy *`g`* incrementally. We have to allocate resources and generate initialization, main loop, and wrapup code. It is used to generate code for the galaxy inside a dynamic construct. A dynamic construct is a wormhole in the code generation domain. When we call the `go` method of the wormhole, we generate code for the inside galaxy.

`virtual int compileCode();`
`virtual int loadCode();`
`virtual int runCode();`
These methods compile and load the code, and run the target. The base class, generates error messages.

`void writeFiring(Star&` *`s`*, int *`depth`*);
This method generates code for a firing of the argument star. The base class simply executes `run` of the star.

`void genLoopInit(Star&` *`s`*, int *`reps`*);
`void genLoopEnd(Star&` *`s`*);
In case loop scheduling is taken, we may want to perform loop initialization routines for stars inside each loop. These methods call `beginLoop` and `endLoop` methods of the argument star.

`void copySchedule(SDFSchedule&` *`sched`*);
If this is a child target, the schedule is inherited from the parallel scheduling of the parent target. This method copies the argument schedule to the schedule of this target and set `noSchedule` flag.

`virtual int systemCall(const char*` *`cmd`*, `const char*` *`error`* = NULL, `const char*` *`host`* `="localhost");`
This method makes a system call using `rshSystem` utility function. If *`error`* is specified and the system call is unsuccessful, display the error message.

`void amInherited();`
This method declares that this target is inherited from other targets.

`virtual int support(Star*` *`s`*);
Returns TRUE if this target allows the argument star; returns FALSE otherwise.

`virtual int execTime(DataFlowStar*` *`s`*, `CGTarget*` *`t`* `= 0);`
We return the execution time of the argument star *`s`* in the argument target *`t`*. In a heterogeneous system, execution time of a given star may vary depending on which target executes the star. In this base class, we just call `myExecTime` method of the star.

17.1.3 Class HLLTarget

Class HLLTarget, derived from CGTarget class, is a base class of all high level language code generation targets. There is AsmTarget class for the base target of all assembly code generation targets. Since we will illustrate the C code generation target, we will explain the HLLTarget class only in this subsection.

HLLTarget class has a constructor with three arguments as CGTarget class. In this base class, we provide some methods to generate C++ code. The following three protected methods are defined to create a C++ identifier, derived from the actual name.

`StringList sanitize(const char* `*`s`*`) const;`
`StringList sanitizedName(const NamedObj& `*`b`*`) const;`
`virtual StringList sanitizedFullName(const NamedObj& `*`b`*`) const;`

> The first method takes a string argument and modifies it with a valid C++ identifier. If the string contains a non-alphanumeric character, it will replace it with ’_’. If the string starts with a number, it prepends ’x’ at the beginning. The second method calls the first method with the name of the argument object. The third method generates an identifier for the argument object that will be placed in `struct` data structure. Therefore, we put ’.’ between the object name and its parent name.

Some public methods are defined.

`void beginIteration(int `*`repetitions`*`, int `*`depth`*`);`
`void endIteration(int `*`repetitions`*`, int `*`depth`*`);`

> If the *`repetitions`* is negative, we print a `while` loop with infinite repetition. Otherwise, we generate a `for` loop. The second argument *`depth`* determines the amount of indent we put in front of the code.

`void wrapup();`

> Saves the generated code to "code.output" file name.

Since this target is not an actual target, it has a pure virtual method: `makeNew.`

17.2 Multiprocessor Targets

There are two base multiprocessor targets: MultiTarget and CGMultiTarget. Class MultiTarget, derived from class CGTarget, serves a base multiprocessor target for CG domain. On the other hand, CGMultiTarget class is the base multiprocessor target for CG domain, thus derived from MultiTarget class. Since the MultiTarget class is a pure virtual class, the derived classes should redefine the pure virtual methods of the class.

Some members only meaningful for CG domain are split to MultiTarget class and the CGMultiTarget class. If they are accessed from the parallel scheduler, some members are placed in MultiTarget class. Otherwise, they are placed in CGMultiTarget class (Note that this is the organization issue). Refer to the CGMultiTarget class for detailed descriptions.

17.2.1 Class MultiTarget

Class MultiTarget, derived from CGTarget, has a constructor with three arguments.

`MultiTarget(const char* name, const char* starclass, const char* desc);`

The arguments are the name of the target, the star class it supports, and the description text. The constructor hides `loopingLevel` parameter inherited from the CGTarget class since the parallel scheduler does no looping as of now.

`IntState nprocs;`

This protected variable (or state) represents the number of processors. We can set this state, and also change the initial value, via the following public method:

`void setTargets(int num);`

After child targets are created, the number of child targets is stored in the following protected member:

`int nChildrenAlloc;`

There are three states, which are all protected, to choose a scheduling option.

```
IntState manualAssignment;
IntState oneStarOneProc;
IntState adjustSchedule;
```

If the first state is set to YES, we assign stars manually by setting `procId` state of all stars. If `oneStarOneProc` is set to YES, the parallel scheduler puts all invocations of a star into the same processor. Note that if manual scheduling is chosen, `oneStarOneProc` is automatically set YES. The last state, `adjustSchedule,` will be used to override the scheduling result manually. This feature has not been implemented yet. There are some public methods related to these states:

```
int assignManually();
int getOSOPreq();
int overrideSchedule();
void setOSOPreq(int i);
```

The first three methods query the current value of the states. The last method sets the current value of the `oneStarOneProc` state to the argument value.

There are two other states that are protected:

```
IntState sendTime;
IntState inheritProcessors;
```

The first state indicates the communication cost to send a unit sample between nearest neighbor processors. If `inheritProcessors` is set to YES, we inherit the child targets from somewhere else by the following method.

`int inheritChildTargets(Target* mtarget);`

This is a public method to inherit child targets from the argument target. If the number of processors is greater than the number of child targets of *`mtarget,`* this method returns FALSE with error message. Otherwise, it copies the pointer to the child targets of *`mtarget`* as its child targets. If the number of processors is 1, we can use a single processor target as the argument. In this case, the argument target becomes the child target of this target.

```
void enforceInheritance();
int inherited();
```

The first method sets the initial value of the `inheritProcessors` state while the second method gets the current value of the state.

```
void initState();
```

Is a redefined public method to initialize the state and implements the precedence relation between states.

Other MultiTarget public members

```
virtual DataFlowStar* createSpread() = 0;
virtual DataFlowStar* createCollect() = 0;
virtual DataFlowStar* createReceive(int from, int to, int num) = 0;
virtual DataFlowStar* createSend(int from, int to, int num) = 0;
```

These methods are pure virtual methods to create Spread, Collect, Receive, and Send stars that are required for sub-universe generation. The last two method need three arguments to tell the source and the destination processors as well as the sample rate.

```
virtual void pairSendReceive(DataFlowStar* snd, DataFlowStar* rcv);
```

This method pairs a Send, *snd,* and a Receive, *rcv,* stars. In this base class, it does nothing.

```
virtual IntArray* candidateProcs(ParProcessors* procs, DataFlowStar* s);
```

This method returns the array of candidate processors which can schedule the star *s.* The first argument is the current ParProcessors that tries to schedule the star . This class does nothing and returns NULL.

```
virtual Profile* manualSchedule(int count);
```

This method is used when this target is inside a wormhole. This method determines the processor assignments of the Profile manually. The argument indicates the number of invocations of the wormhole.

```
virtual void saveCommPattern();
virtual void restoreCommPattern();
virtual void clearCommPattern();
```

These methods are used to manage the communication resources. This base class does nothing. The first method saves the current resource schedule, while the second method restores the saved schedule. The last method clears the resource schedule.

```
virtual int scheduleComm(ParNode* node, int when, int limit = 0);
```

This method schedules the argument communication node, *node,* available at *when.* If the target can not schedule the node until *limit,* return -1. If it can, return the schedule time. In this base class, just return the second argument, *when,* indicating that the node is scheduled immediately after it is available to model a fully-connected interconnection of processors.

`virtual ParNode* backComm(ParNode*` *`node`*`);`

For a given communication node, find a communication node scheduled just before the argument node on the same communication resource. In this base class, return NULL.

```
virtual void prepareSchedule();
virtual void prepareCodeGen();
```

These two methods are called just before scheduling starts, and just before code generation starts, to do necessary tasks in the target class. They do nothing in this base class.

17.2.2 Class CGMultiTarget

While class CGMultiTarget is the base multiprocessor target for all code generation domains, either homogeneous or heterogeneous, it models a fully-connected multiprocessor target. In the target list in pigi, "FullyConnected" target refers to this target. It is defined in `$PTOLEMY/src/domains/cg/targets` directory. It has a constructor with three argument like its base class, MultiTarget.

To specify child targets, this class has the following three states.

```
StringArrayState childType;
StringArrayState resources;
IntArrayState relTimeScales;
```

The above states are all protected. The first state, `childType,` specifies the names of the child targets as a list of strings separated by a space. If the number of strings is fewer than the number of processors specified by `nproc` parameter, the last entry of `childType` is extended to the remaining processors. For example, if we set `nproc` equal to 4 and `childType` to be "default-CG56[2] default-CG96", then the first two child targets become "default-CG56" and the next two child targets become "default-CG96".

The second state, `resources,` specifies special resources for child targets. If we say "0 XXX ; 3 YYY", the first child target (index 0) has XXX resource and the fourth child (index 3) has YYY resource. Here ';' is a delimeter. If a child target (index 0) has a `resources` state already, XXX resource is appended to the state at the end. Note that we can not edit the states of child targets in the current pigi. If a star needs a special resource, the star designer should define `resources` StringArrayState in the definition of the star. For example, a star S is created with `resources` = YYY. Then, the star will be scheduled to the fourth child. One special resource is the target index. If `resources` state of a star is set to "2", the star is scheduled to the third target (index 2).

The third state indicates the relative computing speed of the processors. The number of entries in this state should be equal to the number of entries in `childType.` Since we specify the execution of a star with the number of cycles in the target for which the star is defined, we have to compensate the relative cycle time of processors in case of a heterogeneous target environment.

Once we specify the child targets, we select a scheduler with appropriate options. States inherited from class MultiTarget are used to select the appropriate scheduling options. In the CGMultiTarget class, we have the following three states, all protected, to choose a scheduler unless the manual scheduling option is taken.

```
IntState ignoreIPC;
IntState overlapComm;
IntState useCluster;
```

The first state indicates whether we want to ignore communication overhead in scheduling or not. If it says YES, we select the Hu's Level Scheduler . If it says NO, we use the next state, `overlapComm.` If this state says YES, we use the dynamic level scheduler . If it says No, we use the last state, `useCluster.` If it says YES, we use the declustering algorithm . If it says NO, we again use the dynamic level scheduler. By default, we use the dynamic level scheduler by setting all states NO. Currently, we do not allow communication to be overlapped with computation. If more scheduling algorithms are implemented, we may need to introduce more parameters to choose those algorithms.

There are other states that are also protected.

```
StringState filePrefix;
```

Indicates the prefix of the file name generated for each processor. By default, it is set to "code_proc", thus creating code_proc0, code_proc1, etc for code files of child targets.

```
IntState ganttChart;
```

If this state says YES (default), we display the Gantt chart of the scheduling result.

```
StringState logFile;
```

Specifies the log file.

```
IntState amortizedComm;
```

If this state is set to YES, we provide the necessary facilities to packetize samples for communication to reduce the communication overhead. These have not been used nor tested yet.

Now, we discuss the three basic methods: `setup,` run, wrapup.

```
void setup();
```

(1) Based on the states, we create child targets and set them up: `prepareChildren.`

```
virtual void prepareChildren();
```

This method is protected. If the children are inherited, it does nothing. Otherwise, it clears the list of current child targets if they exist. Then, it creates new child targets by `createChild` method and give them a unique name using `filePrefix` followed by the target index. This method also adjusts the `resources` parameter of child targets with the `resources` specified in this target: `resourceInfo.` Finally, it initializes all child targets.

`virtual Target* createChild(int` *`index`*`);`

This protected method creates a child target, determined by `childTypes,` by *`index.`*

```
virtual void resourceInfo();
```

This method parses the `resources` state of this class and adjusts the `resources` parameter of child targets. If no `resources` parameter exists in a child target, it creates one.

(2) Choose a scheduler based on the states: `chooseScheduler.`

```
virtual void chooseScheduler();
```

This is a protected method to choose a scheduler based on the states related to scheduling algorithms.

(3) If it is a heterogeneous target, we flatten the wormholes: `flattenWorm.` To represent a universe for heterogeneous targets, we manually partition the stars using wormholes: which stars are assigned to which target.

```
void flattenWorm();
```

This method flattens wormholes recursively if the wormholes have a code generation domain inside.

(4) Set up the scheduler object. Clear `myCode` stream.

(5) Initialize the flattened galaxy, and perform the parallel scheduling: `Target::setup.`

(6) If the child targets are not inherited, display the Gantt chart if requested: `writeSchedule.`

```
void writeSchedule();
```

This public method displays a Gantt chart.

(7) If this target is inside a wormhole, it adjusts the sample rate of the wormhole ports (`CGTarget::adjustSampleRates),` generates code (`generateCode),` and downloads and runs code in the target (`CGTarget::wormLoadCode).`

```
void generateCode();
```

This is a redefined public method. If the number or processors is 1, just call `generateCode` of the child target and return. Otherwise, we first set the stop time, or the number of iteration, for child targets (`beginIteration).` If the target is inside a wormhole, the stop time becomes -1 indicating it is an infinite loop. The next step is to generate wormhole interface code (`wormInputCode,` wormOutCode if the target is inside a wormhole. Finally, we generate code for all child targets (`ParScheduler::compileRun).` Note that we generate wormhole interface code before generating code for child targets since we can not intervene the code generation procedure of each child target once started.

`void beginIteration(int` *`repetitions`*, `int` *`depth`*);
`void endIteration(int` *`repetitions`*, `int` *`depth`*);

These are redefined protected methods. In the first method, we call `setStopTime` to set up the stop time of child targets. We do nothing in the second method.

`void setStopTime(double` *`val`*);

This method sets the stop time of the current target. If the child targets are not inherited, it also sets the stop time of the child targets.

```
void wormInputCode();
void wormOutputCode();
```
`void wormInputCode(PortHole&` *`p`*);

`void wormOutputCode(PortHole& p);`

These are all redefined public methods. The first two methods traverse the portholes of wormholes in the original graph, find out all portholes in sub-universes matched to each wormhole porthole, and generate wormhole interface code for the portholes. The complicated thing is that more than one ParNode is associated with a star and these ParNodes may be assigned to several processors. The last two methods are used when the number of processors is 1 since we then use `CGTarget::wormInputCode,wormOutputCode` instead of the first two methods.

`int run();`

If this target does not lie in a wormhole or it has only one processor, we just use `CGTarget::run` to generate code. Otherwise, we transfer data samples to and from the target: `sendWormData` and `receiveWormData.`

`int sendWormData();`
`int receiveWormData();`

These are redefined protected methods. They send data samples to the current target and receive data samples from the current target. We traverse the wormhole portholes to identify all portholes in the sub-universes corresponding to them, and call `sendWormData,` receiveWormData for them.

`void wrapup();`

In this base class, we write code for each processor to a file.

Other CGMultiTarget protected members

`ParProcessors* parProcs;`

This is a pointer to the actual scheduling object associated with the current parallel scheduler.

`IntArray canProcs;`

This is an integer array to be used in `candidateProcs` to contain the list of processor indices.

`virtual void resetResources();`

This method clears the resources this target maintains such as communication resources.

`void updataRM(int from, int to);`

This method updates a reachability matrix for communication amortization. A reachability matrix is created if `amortizedComm` is set to YES. We can packetize communication samples only when packetizing does not introduce deadlock of the graph. To detect the deadlock condition, we conceptually cluster the nodes assigned to the same processors. If the resulting graph is acyclic, we can packetize communication samples. Instead of clustering the graph, we set up the reachability matrix and update it in all send nodes. If there is a cycle of send nodes, we can see the deadlock possibility.

Other CGMultiTarget public members

The destructor deletes the child targets, scheduler, and reachability matrix if they exist. There is an `isA` method defined for type identification.

`Block* makeNew() const;`

Creates an object of CGMultiTarget class.

`int execTime(DataFlowStar*` *`s`*`, CGTarget*` *`t`*`);`

This method returns the execution time of a star *`s`* if scheduled on the given target *`t`*. If the target does not support the star, a value of -1 is returned. If it is a heterogeneous target, we consider the relative time scale of processors. If the second argument is NULL or it is a homogeneous multiprocessor target, just return the execution time of the star in its definition.

`IntArray* candidateProcs(ParProcessors*` *`par`*`, DataFlowStar*` *`s`*`);`

This method returns a pointer to an integer array of processor indices. We search the processors that can schedule the argument star *`s`* by checking the star type and the resource requirements. We include at most one idle processor.

`int commTime(int` *`from`*`, int` *`to`*`, int` *`nSamples`*`, int` *`type`*`);`

This method returns the expected communication overhead when transferring *`nSamples`* data from *`from`* processor to *`to`* processor. If *`type`* = 2, this method returns the sum of receiving and sending overhead.

`int scheduleComm(ParNode*` *`comm`*`, int` *`when`*`, int` *`limit`* `= 0);`

Since it models a fully-connected multiprocessor, we can schedule a communication star anytime without resource conflict that returns the second argument *`when`*.

`ParNode* backComm(ParNode*` *`rcv`*`);`

This method returns the corresponding send node paired with the argument receive node, *`rcv`*. If the argument node is not a receive node, return NULL.

`int amortize(int` *`from`*`, int` *`to`*`);`

This method returns TRUE or FALSE, based on whether communication can be amortized between two argument processors.

17.2.3 Class CGSharedBus

Class CGSharedBus, derived from class CGMultiTarget, is a base class for shared bus multiprocessor targets. It has the same kind of constructor as its base class.

This class has an object to model the shared bus.

```
UniProcessor bus;
UniProcessor bestBus;
```

These are two protected members to save the current bus schedule and the best bus schedule obtained so far. The `bus` and `bestBus` are copied to each other by the following public methods.

`void saveCommPattern();`

`void restoreCommPattern();`
`clearCommPattern();`
`void resetResources()`

The first method is a public method to clear `bus` schedule, while the second is a protected method to clear both `bus` and `bestBus.`

This classes redefines the following two public methods.

`int scheduleComm(ParNode*` *`node`*`, int` *`when`*`, int` *`limit`* `= 0);`

This method schedules the argument node available at *`when`* on `bus.` If we can schedule the node before *`limit,`* we schedule the node and return the schedule time. Otherwise, we return -1. If *`limit`* = 0, there is no limit on when to schedule the node.

`ParNode* backComm(ParNode*` *`node`*);

For a given communication node, find another node scheduled just before the argument node on `bus.`

17.3 Heterogeneous Support

In this section, we summarize the special routines to support heterogeneous targets. They are already explained in earlier chapters.

1. To specify the component targets , we first set `childTypes` state of the target class that must be derived from class CGMultiTarget. We may add special resources to the processors by setting `resources` state, a list of items separated by ';'. An item starts with the target index followed by a list of strings identifying resources. The relative computing speed of processors are specified by `relTimeScales` state.
2. An application program for a heterogeneous target uses wormholes. In pigi, all stars in a universe should be in the same domain. To overcome this restriction, we use wormhole representation to distinguish stars for different targets, or domains, but still in the same universe. Once the graph is read into the Ptolemy kernel, all wormholes of code generation domain are flattened to make a single universe: `flattenWorm` method of CGMultiTarget class. Currently, we manually partition the stars to different kinds of processors. For example, if we have three "default-CG96" targets and one "default-CG56" target, we partition the stars to two kinds: CG96 or CG56. This partitioning is based on the original wormhole representation. If we ignore this partitioning, we can apply an automatic scheduling with the flattened graph. This feature has not been tested yet even though no significant change is required in the current code.
3. When we schedule a star in the scheduling phase, we first obtain the list of processors that can schedule the star: `candidateProcs` method of `CGMultiTarget` class. The execution time of the star to a processor is computed in `execTime` method of `CGMultiTarget` class considering the relative speed of processors.

4. After scheduling is performed, we create sub-universes for child targets. In case manual partitioning is performed, we just clone the stars from the original graph in the sub-universes. In case we use automatic partitioning, we need to create a star in the current target with the same name as the corresponding star in the original graph: `cloneStar` private method of `UniProcessor` class. We assume that we use the same name for a star in all domains.

Chapter 18. CGC Domain

Authors: *Soonhoi Ha*

Other Contributors: *Mudit Goel*

In this chapter, we will explain the current implementation of C code generation domain. The source code can be found `$PTOLEMY/src/domains/cgc/kernel` directory. We follow the general framework for code generation defined in CG kernel directory.

In the CGC domain, the resource we have is the name space. We have to avoid name conflicts by guaranteeing unique names for different variables. The most complicated task is to determine the dimension, or buffer size, of each variable, and the method how to access them; static buffering, linear indexing, or modulo addressing.

We use the CGC domain to test new functionalities in code generation: buffer embedding for example. We have tested some simple demos to verify the design.

18.1 Buffer Allocation

In the CGC domain, we allocate one buffer for each connection in principle. We have to determine the required size of buffers first. If a porthole is *embedded* , and the buffer size requirement is equal to the sample rate of the embedded port, we do NOT allocate a buffer on that connection. We will use static buffering for all *embedded* and *embedding* portholes. If the buffer requirement of an embedded(or embedding) porthole is not equal to the sample rate of the porthole, we actually need to have two buffers on that connection and copy data between these buffers. In this case, we splice a Copy star on the arc and schedule the Copy star appropriately to generate code for copying data. After inserting the Copy star, we will end up with one buffer per connection. Another cause of copy requirement is type conversion from complex to float/int or from float/int to complex. Then, we splice a type-conversion star on the arc.

Class CGCTarget redefines the following protected method for buffer allocation .

```
int allocateMemory();
```

In this method, we first merge cascaded forks into a single fork whose input keeps the list of all fork destinations. We will allocate only one buffer for each fork. All fork destinations will refer to the same fork input buffer. Then this method does the following tasks:

1. Determine the buffer requirements for all portholes.
2. Splice Copy stars or type conversion stars if necessary.
3. Set the buffer type for each output porthole: either OWNER or EMBEDDED. If the output porthole is embedded, or the corresponding input porthole is embedded, it is called EMBEDDED. Otherwise, it is OWNER. The buffer type of an output is determined using the following public method of CGCPortHole class:

```
void setBufferType();
```

4. We assign unique names for buffers.
5. We initialize the offset pointer for each porthole which is associated with a buffer of size greater than 0 (`initOffset` method of CGCPortHole class). This offset pointer indicates from which offset of the buffer the porthole starts reading or writing samples.

`int initOffset();`

This is a public method of CGCPortHole class to initialize the offset pointer. If there are delays, or initial samples, on the arc, these samples are placed at the end of the buffer. The offset pointer of a porthole indicates the location of the last sample the next firing of its parent star will produce or consume. It is compatible with the SDF simulation domain: `$ref(porthole,num)` in CGC stars is now equivalent to `porthole%num` in SDF stars. We can set the offset pointer of an output porthole manually by the following public method of CGCPortHole class.

`void setOffset(int` *`v`*`);`

Now, we will explain steps (1), (2), and (4) in more detail.

18.1.1 Buffer requirement

To determine the buffer requirements of portholes, we traverse portholes of all stars, and call `finalBufSize` method of CGCPortHole class.

`void finalBufSize(int` *`statBuf`*`);`

This is a public method of CGCPortHole class to determine the buffer size for this porthole. The argument indicates whether we try to use static buffering or not. We allocate one buffer for each connection. Therefore, we do nothing if this porthole is an input porthole. If this porthole is disconnected, we set the buffer size equal to the number of samples produced for each firing. If it lies at wormhole boundary, we use `localBufSize` method of CGPortHole class to determine the size of buffer and return. Otherwise, we do the following:

1. We can manually assign the buffer size by calling `requestBufSize` for an output porthole of interest in the setup stage of a star:

`void requestBufSize(int` *`sz`*`);`

This method sets the buffer size manually. The argument size should not be smaller than the minimum size determined by the scheduler. The minimum size determined by the scheduler is the sum of maximum number of samples accumulated on the arc during the schedule and the number of old samples to be access from the destination port. If *`sz`* is smaller than this minimum value, we generate a warning message and give up manual allocation.

2. We set the initial buffer size by calling `localBufSize` method of CGPortHole class. If argument *`statBuf`* = 1, we set the buffer size as a smallest multiple of the sample rate of this porthole, which is not less than the initial buffer size. By doing this, we increase the chance of using linear buffering. We also set the waste factor in CGCGeodesic class to a huge number by calling the following public method in

CGCGeodesic class:

`void preferLinearBuf(int i)`

The waste factor set by the above method can be obtained by the following redefined protected method of the CGCGeodesic class.

```
double wasteFactor() const;
```

3. We set two flags for this porthole to indicate we can use static buffering and/or linear buffering: `hasStaticBuf` and `asLinearBuf.` These two flags are all private. If static buffering flag is set, we use direct addressing in the generated code to access the buffer. If linear buffering flag is set, we will use indirect addressing and no modulo addressing will be required. Otherwise, we will use indirect addressing and modulo addressing in the generated code to access the allocated buffer. Initially both flags are set TRUE. If this porthole needs to access past samples, we reset both flags to FALSE. When the argument *`statBuf`* is given 0, we give up static buffering in case the buffer size determined in (2) is greater than the sample rate of this porthole. Note that if a loop scheduler is used, *`statBuf`* becomes 0 and some possibilities of static buffering are sacrificed as the cost of code compaction. The following method is called to adjust the flags further.

```
void setFlags();
```

Is a protected member of CGCPortHole class. If the final buffer size is not a multiple of the sample rate, we reset `asLinearBuf` flag to 0. We have to use modulo addressing in the generated code. If the product of the sample rate and the repetition count of its parent star is not a multiple of the final buffer size, we give up static buffering, setting `hasStaticBuf` to 0. If an output porthole is embedded or embedding, we set both flags TRUE since we enforce static buffering.

4. As the final step, we set the flags for destination portholes. If this porthole is connected to a fork input, all fork destinations will be the destination portholes of this porthole. We first check whether *`statBuf`* argument is 0 and the buffer size is greater than the sample rate of the porthole. And, we call `setFlags` method for that porthole. If the porthole needs to access past samples, or the number of initial samples on the connection is not a multiple of the sample rate, we give up linear buffering.

The final buffer size can be obtained by the following two public methods of CGCPortHole class.

```
int maxBufReq() const;
int bufSize() const { return maxBufReq(); }
```

The above methods return the final buffer size associated with this porthole. If it is a fork destination, it returns the size of the fork input buffer. If the porthole has switched its Geodesic , it returns the size of buffer associated with the switched Geodesic.

The flags for static buffering and linear buffering can be obtained by the following public methods of CGCPortHole class:

```
int linearBuf() const;
int staticBuf() const;
```

We give up static buffering for a CGPortHole by calling the following public method of CGCPortHole class.

```
void giveUpStatic();
```

18.1.2 Splice stars

After buffer requirements for all portholes are determined, we can detect the arcs which can not have only one buffer. For instance, if we need to convert data types from complex to float/int or vice versa automatically, we need two buffers on the arc: one for complex variables and the other for float/int variables. This copying operation is required since C language does not provide built-in "complex" type variable. Therefore, we define "complex" type data in the generated code as follows;

```
static char* complexDecl =
"\back n#if !defined(COMPLEX_DATA)\back n#define COMPLEX_DATA 1"
"\back n typedef struct complex_data { double real; double imag; }
complex; \back n"
"#endif\back n";
```

Another case is when an embedded or embedding porthole requires a buffer whose size is greater than the sample rate of the porthole. Recall that an embedded or embedding porthole will assume static buffering for each execution when we generate code for that porthole. If the buffer size is larger than the sample rate, we may not use static buffering. We need two buffers for the embedded or embedding porthole.

Rather than assigning two buffers on an arc and letting the target generating code to copy data between these two buffers, we splice a star on the arc. The spliced star will separate two buffers on one arc into one buffer on its input and the other buffer on its output arc. When this spliced star is scheduled before the destination star of after the source star, it will generate code to copy data from the input buffer to the output buffer.

Stars are spliced in the following protected method of CGCTarget class.

```
void addSpliceStars();
```

This method traverses all portholes of stars in the galaxy.

When we splice a star at an input porthole (destination porthole), we initialize the spliced star and set the target pointer. A spliced star should have one "input" and one "output". We set the sample rate of these portholes equal to the sample rate of the input porthole. The buffer size of the input porthole of the spliced star is determined by the original source porthole. The buffer size of the output porthole is set the sample rate of the input porthole. And, we check whether static or linear buffering can be used for the portholes. The input porthole of a spliced Copy star gives up static buffering while the output porthole of the spliced Copy star and the original destination porthole can use static and linear buffering. In case we spliced a type conversion star, we need to change the type of the original source porthole.

We splice a Copy star at the output (source porthole) when the output is an embedded or embedding porthole and the buffer size is larger than the sample rate of the output porthole. We initialize the spliced star and set the target pointer. The sample rate of the input and output porthole of the spliced Copy star is equal to the sample rate of the output porthole. The buffer size of the output porthole of the spliced star is set to the buffer size of the arc. We give up static buffering for this output porthole. On the other hand, we change the buffer size of the source porthole to the sample rate of the porthole.

We need to pay special attention to Collect (or Spread) stars. A Collect (or Spread) star is not a regular SDF star so that it is not scheduled when all input data are available. Actually, we do not execute the spliced Collect (or Spread) stars. But, the output porthole of a Collect (or Spread) star is an embedding (or embedded) porthole. And its buffer size can be larger than the sample rate of the porthole. In this case, we splice a Copy star at the destination porthole, not at the source porthole. We schedule this Copy star before the destination star. The sample rate of portholes of the spliced Copy star is equal to the sample rate of the destination porthole. The output buffer size of the spliced star is set the the buffer size of the arc while the input buffer size now becomes the sample rate of the source porthole. The trickest part here is to determine the offset pointers. We copy data when the destination porthole requires it. Therefore, the offset pointers of the input porthole and the output porthole of the spliced Copy star depends on the initial delay on the arc. We manually set the offset by `setOffset` method of CGCPortHole class.

There is another case we need data copying between two buffers: when two embedded portholes are connected together. Suppose, an output porthole of a Spread star is connected to an input porthole of a Collect star. Since the output porthole of a Spread star is embedded to the input buffer and the input porthole of a Collect star is embedded to the output buffer, we need to copy data from the input buffer of the Spread star to the output buffer of the Collect star. Since we do not schedule neither Spread nor Collect star, we may not splice a Copy star either at the source porthole not at the destination porthole. Therefore, we leave it as a special case so that we generate code to copy data between two buffers in `moveDataBetweenShared` method of CGCStar class after executing the star connected to the input porthole of the Spread star. So, we do not splice star when two embedded portholes are connected together.

```
void moveDataBetweenShared();
```

This is a protected method of CGCStar class. This method is called inside `runIt` method after generating code for a star. If the star is connected to an embedding porthole of a star of which an embedded output porthole is connected to an embedded porthole. Since we meet the case when two embedded portholes are connected, we generate code for copying data between two embedding buffers.

Scheduling spliced stars

When we splice a star at the input port of a star, we want to schedule the spliced star before the star. On the other hand, we want to schedule the spliced star after a star if we splice a star at the output porthole of the star. When we splice stars, we are already given the schedule. Therefore, we need to insert spliced stars into the schedule. An intuitive approach is to insert them into the schedule list.

Currently, we use a simpler method. We use the fact that the spliced star and the star connected to the spliced star can be regarded as a cluster and schedule of that cluster is well known. Our idea is to actually execute the cluster when we execute a star if the star is connected to spliced stars. CGCStar class has a private member to keep the list of stars: `spliceClust.` Initially, the star itself is inserted to the list. If we splice a star at the input porthole, we prepend the spliced star to the list. If we splice a star at the output porthole, we append the spliced star to the list. And, we redefine `run` method.

`int run();`

If there are spliced stars, or the list size is greater than 1, we traverse the list and execute `runIt` method for each star. Otherwise, we execute `runIt` method.

`int runIt();`

It is a protected method of CGCStar class to generate main code for this star. If generates a comment regarding this star and main code. It updates offset pointers of the star. Finally, it calls `moveDataBetweenShared` method to generate code to copy data between two embedding portholes if necessary.

18.1.3 Buffer naming

One major task for resource assignment in the CGC domain is to give a unique name for each variable. In the setup stage of the CGCTarget, we assign a unique index value to each star starting from 1 to the number of stars in the galaxy. The CGCTarget has two protected members to give a unique index for galaxy.

```
int galId;
int curId;
```

The second member is used to give unique indices for galaxies while the first member indicates the index of the current galaxy.

Now, the CGCTarget can generate a unique name for each variable, portholes and states, by the following protected method.

`StringList sanitizedFullName(const NamedObj& `*b*`) const;`

In this method, the argument object is a porthole or a state of a star. We prefix ’g’ followed by the galaxy index, followed by "_", followed by the name of the star, followed by another ’_’, followed by the star index, followed by yet another ’_’ to the name of the object. For example, if star A has a state xx and the star index is 2 and the galaxy index is 1, the name of the state becomes "g1_A_2_xx".

`StringList correctName(const NamedObj& `*b*`);`

Is a public version of `sanitizedFullName` method.

Now, we are ready to generate unique names for portholes.

`void setGeoName(char* `*name*`);`

Is a public method of CGCPortHole class. If this porthole is disconnected and no Geodesic is assigned, we store the name in the porthole. Otherwise, we store the name in the Geodesic by calling the following public method of CGCGeodesic class.

`void setBufName(char* `*name*`);`

The buffer name of a porthole can be obtained by the following public method of CGCPortHole class.

`const char* getGeoName() const;`

This method returns the buffer name stored in this object if it is disconnected, or call `getBufName` method of CGCGeodesic class. If it is a fork destination, it returns the name of the fork input buffer.

18.2 Data structure for galaxy and stars

In the global declaration section of the generated code, we declare data structures for stars. At early design stage of CGC domain, we use `struct` construct of C language to declare the data structure of the program. This way, we could assign unique memory locations to variables very easily. But, the length of a variable gets large as the hierarchy of the graph grows. Furthermore, we reduce significant amount of compiler optimization possibility. Therefore, we invented a scheme to generate unique symbols for variables (`sanitizedFullName` of CGCTarget class) without using "struct" construct.

```
virtual void galDataStruct(Galaxy& galaxy, int level = 0);
virtual void starDataStruct(CGCStar* block, int level = 0);
```

The above methods are protected methods of CGCTarget class to be called in `frameCode` method to declare data structures of galaxy and stars. The second argument of both methods indicates the depth of hierarchy which the first argument block resides in, thus advising the amount of indents in the generated code. By default, it is set 0. The first method calls the second method for each component star if it is not a Fork star. We do not generate code nor declare data structure for Fork stars.

The data structure for a star consists of four fields:

1. Comments to indicate that the following declarations corresponds to what star: `sectionComment` method.

```
StringList sectionComment(const char* string);
```

This is a protected method of CGCTarget class to generate a comment line, *`string`* in the generated code.

2. Declare buffers associated with portholes. We do not declare input portholes. If an output porthole is EMBEDDED, we declare a pointer to the embedding buffer, by prepending ’*’ in front of the buffer name. Otherwise, it declare a regular buffer.

3. Declare index pointers to the buffer if static buffering is not used and the size of buffer is greater than 1 . Portholes will use these index pointers to locate the buffer position. For a regular buffer, we declare an index pointer, named after the buffer name appended by "_ix". The name of index porthole is given by `offsetName` method of CGCTarget class.

```
StringList offsetName(const CGCPortHole* p);
```

This is a public method to assign an index pointer to the argument porthole. It appends ’_’ followed by "ix" at the end of the porthole name, by calling the following public method of CGCTarget class:

```
StringList appendedName(const NamedObj& p, const char* add);
```

This method is used to append ’_’ followed by *`add`* to the name of the object *`p`*.

4. Finally, we declare referenced states. A State is called *referenced* only when we

use $ref macro for the state at most once. CGCStar class has the following members for referenced states:

```
StateList referencedStates;
void registerState(const char* name);
```

The first is a public member to store the list of referenced states in this Star. The second is a protected method to add the state with given name to the list of referenced states if not inserted.

We traverse the list of referenced states to declare variables. Unlike portholes, the size of a state variable is given. If the size of state is 1, we both declare and initialize the state. If the state is an array state, we both declare and initialize the state using array initialization unless the state is declared inside a function. If we declare an array state inside a function, we have to write explicit initialization code. Class CGCTarget has the following public method to tell whether we are working inside a function or not.

```
int makingFunc();
```

Returns TRUE if we are defining a function.

18.2.1 Buffer initialization

We initialize buffers and index pointers as follows.

1. If the buffer is EMBEDDED, we assign a pointer to the embedded buffer and set the pointer to the starting address of the embedding buffer, from which the buffer is embedded. If the size of the embedding buffer is 1, we assign the pointer of the embedding buffer.
2. For the regular buffer, we initialize with 0s in case the buffer size is greater than 1.
3. We initialize an index pointer of a buffer to the offset pointer of the porthole associated with that index pointer.

18.3 CGC code streams

Besides two code streams inherited from CGTarget class, `myCode` and `procedures`, CGCTarget class maintains 9 more code streams (all protected). These code streams will be stitched together to make the final code in `frameCode` method. There are two schemes to organize a code in general. One scheme would be to put code strings to a single CodeStream in order. For example, we put global declarations, main function declaration, initialization, and main loop into a single `myCode` stream in order. For single processor code generation, it would be feasible. For multiprocessor case, however, the parent target may add some extra code strings. Therefore, we assign different code streams to different section of code. On the other hand, if we have too many code streams, it would be arduous to remember all.

```
CodeStream globalDecls;
CodeStream galStruct;
CodeStream include;
```

These three code streams will be placed in the global scope of the final code. The galaxy declaration (`galStruct`) is separated from `globalDecls` because we need to put galaxy declaration inside a function if we want to define a function from a galaxy (for example, recursion construct). A programmer can provide strings to `globalDecls` and `include` by using the following protected CGCStar methods in a star definition:

```
int addGlobal(const char* decl, const char* name = NULL);
int addInclude(const char* decl);
```

In the first method, we use *`decl`* strings as the name if the second argument is given NULL, to make a global declaration unique. The argument of the second method is the name of a file to be included, for example <stream.h> or "DataStruct.h".

```
CodeStream mainDecls;
CodeStream mainInit;
CodeStream commInit;
```

These three code streams will be placed in the main function before the main loop: declaration in the main function, initialization code, and initialization code for communication stars. We separated `commInit` from `mainInit` since communication stars are inserted by the parent multiprocessor target. A programmer can provide strings to the first two code streams by using the following protected CGCStar methods.

```
int addDeclaration(const char* decl, const char* name = NULL);
int addMainInit(const char* decl, const char* name = NULL);
```

The first method uses *`decl`* string as the name of the string if *`name`* is given NULL.

```
CodeStream wormIn;
CodeStream wormOut;
CodeStream mainClose;
```

The first two streams contain code sections to support wormhole interface to the host machine. They will be placed at the beginning of the main loop and at the end of the main loop. The last code stream will be placed after the main loop in the main function.

Recall that using `addCode` method defined in CGStar class, we can put code strings to any code stream .

These nine code streams are initialized by the following protected method of CGCTarget class.:

```
virtual void initCodeStrings();
```

Note that code streams are not initialized in `setup` method of the target since the parent target may put some code before calling the `setup` method of the target. We initialize code streams after we stitch them together and copy the final code in `myCode` stream in `frameCode` method. We do not initialize `myCode` stream in the above method.

```
void frameCode();
```

This method put all code streams together and copy the resulting code to `myCode` stream.

18.4 Other CGCPortHole members

CGCPortHole is derived from CGPortHole class. It has a constructor with no argument. In the constructor, we initialize the default properties of a CGCPortHole: static buffering and linear buffering flags are set TRUE, buffer size is set to 1. These properties are also initialized in `initialize` method. In the destructor, it deallocates the name of the buffer if stored in this class (when this porthole is disconnected). All members described in this section are public.

`CGCPortHole* getForkSrc();`
`const CGCPortHole* getForkSrc() const;`

These methods return the fork input porthole (`forkSrc`) if this porthole is a fork destination. The second method is the *const* version of the first method.

`CGCPortHole* realFarPort();`
`const CGCPortHole* realFarPort() const;`

These method return the far side porthole. If the far side porthole is a fork destination, they return the far side porthole of the fork input, thus bypassing fork stars. The second is the *const* version of the first method.

`CGCGeodesic& geo();`
`const CGCGeodesic& geo() const;`

Return the geodesic connected to this PortHole, type cast. The second is the `const` version of the first method.

`Geodesic* allocateGeodesic();`

Allocates a CGCGeodesic.

`void setupForkDests();`

If this method is called for a fork input porthole, make a complete list of `forkDests` considering all cascaded forks.

`int inBufSize() const;`

This method returns the `bufferSize` of this porthole.

CGCPortHole has an iterator called `ForkDestIter.` It returns fork destinations one at a time. The return type is CGCPortHole.

The derived classes of CGCPortHole in the CGC domain are `InCGCPort, OutCGCPort, MultiCGCPort, MultiInCGCPort,` and `MultiOutCGCPort.`

18.5 Other CGCStar members

Class CGCStar is derived from CGStar class. It has a constructor with no argument. CGCTarget class is a friend class. It has a method to return the domain it lies in (`domain`) and a method for class identification (`isA`). In `initialize` method, we initialize `referencedStates` list. All other members described in this section are all protected.

`CGCTarget* targ();`

Returns the target pointer, type cast to CGCTarget.

`StringList expandRef(const char*` *`name`*`);`
`StringList expandRef(const char*` *`name`*`, const char*` *`offset`*`);`

The above methods resolve macro $ref. The *`name`* argument is a state name or a porthole name. If it is a state name, we put the state in the `referencedStates` list. In the second method, the second argument is the offset of the first argument (state or porthole). It can be a numeral, an IntState name, or a string. If it is an IntState, the current value of the state is taken.

There are various ways to referring to a porthole. If the buffer size is 1, we use the buffer name or the pointer version depending on the type, EMBEDDED or OWNER. If the buffer size is larger than 1, we use direct addressing if static buffering is used. If static buffering can not be used, we use indirect addressing. The following method generates indirect addressing:

`virtual StringList getActualRef(CGCPortHole* p, const char* ix);`

This method generates an indirect addressing for the argument porthole *p* with offset *ix*. If we may not use linear addressing, we generate modulo addressing, in which the index is modulo the buffer size.

`virtual int amISpreadCollect();`

Returns TRUE or FALSE, based on whether this star is a Spread or a Collect star or not. We need to take special care for Spread and Collect stars.

18.6 Other CGCTarget members

CGCTarget is derived from HLLTarget class which is the base target class for high level language code generation. It has a constructor with three argument like its base target classes. In the constructor, we initialize code streams and put them into the `codeStringLists` by `addStream` method. It has `makeNew` method defined.

18.6.1 Other CGCTarget protected members

CGCTarget class has many states guiding the compilation procedure.

`IntState doCompile;`

If this state is set NO, we only generate code, not compiling the code.

```
StringState hostMachine;
StringState funcName;
StringState compileCommand;
StringState compileOptions;
StringState linkOptions;
```

The `hostMachine` state indicates where the generated code is compiled and run. If this state does not indicate the current host,, we will use remove shell command for compilation and execution. The `funcName` state is by default set "main". For multiprocessor code generation case, we may want to give different function name for the generated code. The next three states determines the compilation command:

`compileCommand` `compileOptions` fileName `linkOptions`

There are some other states defined in this class.

`IntState staticBuffering;`

If this state is set YES, we increase the `wasteFactor` of geodesics to use static buffering as much as possible, which is default.

`StringState saveFileName;`

We save the generated code in this file if the file name is given.

`StringArrayState resources;`

This state displays which resources this target has. By default, the CGCTarget has the standard I/O (*`STDIO)`* resource. If a derived target does not support the standard I/O, it should clear this state.

`int codeGenInit();`

This method generates initialization code: buffer initialization, and `initCode` method of all stars. Before generating initialization code, we switch the `myCode` pointer of stars to the `mainInit` code stream so that `addCode` method called inside the `initCode` method puts the string into the `mainInit` code stream.

`void compileRun(SDFScheduler*` *`s`*);

Before calling `compileRun` method of the SDFScheduler, which will call `run` method of stars in the scheduled order, we switch the `myCode` pointer of stars back to the `myCode` code stream of the target. After code generation, we switch the pointer of stars to the `mainClose` code stream for wrapup stage.

`int wormLoadCode();`

If the `doCompile` state is set NO, we just return TRUE, doing nothing. Otherwise, we compile and run the generated code. Return FALSE if any error occurs.

`StringList sectionComment(const char*` *`s`*);

This method makes a comment statement with the given string in C code.

`void wormInputCode(PortHole&` *`p`*);
`void wormOutputCode(PortHole&` *`p`*);

The above methods just print out comments. We haven't supported wormhole interface for CGC domain yet (Sorry!).

18.6.2 Other CGCTarget public members

`void setup();`

This method initialize `galId`, `curId` indices for unique symbol generation. It also generate indices for stars and portholes. Then, it calls `CGTarget :: setup` for normal setup procedure.

`void wrapup();`

This method displays the generated code stored in `myCode` stream. If the galaxy is not inside a wormhole, it calls `wormLoadCode` method to compile and run the code.

`int compileCode();`

This method compiles the generated code. The compile command is generated by the following method:

`virtual StringList compileLine(const char*` *`fName`*);

The argument for this method is the file name to be compiled. If the `hostMachine` does not indicate the local-host, we use remote shell.

`int runCode();`

This method runs the code. If the `hostMachine` is not the local-host, we use `rshSystem` function.

`void headerCode();`

Is redefined to generate a valid C comment with the target name.

`void beginIteration(int` *`repetitions`*, `int` *`depth`*);
`void endIteration(int` *`repetitions`*, `int` *`depth`*);

The first method generates the starting line of `while` loop (if *`repetitions`* is negative) or `for` loop (otherwise). After that it appends the `wormIn` code stream to the `myCode` stream before stars fill the loop body. The `wormIn` code stream is already filled. The second method close the loop. Just before closing the loop, it appends the `wormOut` code stream to the `myCode` at the end of the loop body.

`void setHostName(const char*` *`s`*);
`const char* hostName();`

The above methods set and get the `hostName` state.

`void writeCode(const char*` *`name`* = NULL);

If the argument is NULL, we use the galaxy name as the file name. This method saves the code to the file.

`void wantStaticBuffering();`
`int useStaticBuffering();`

These methods set and get the `staticBuffering` state.

`int incrementalAdd(CGStar*` *`s`*, `int` *`flag`* = 1);

We add the code for the argument star, *`s,`* during code generation step. If *`flag`* is 0, we add the main body of the star (`go` method only). Otherwise, we initialize the star, allocate memory, and generate initialization code, main body, and wrapup code.

`int insertGalaxyCode(Galaxy*` *`g`*, `SDFScheduler*` *`s`*);

We insert the code for the argument galaxy during code generation procedure. We give the unique index for the galaxy and set the indices of stars inside the galaxy. Then, it calls `CGTarget :: insertGalaxyCode` to generate code. After all, we declare the galaxy.

`void putStream(const char*` *`n`*, `CodeStream*` `cs`);
`CodeStream* removeStream(const char*` *`n`*);

The above methods put and remove a code stream named *`n.`*

18.7 Class CGCMultiTarget

Class CGCMultiTarget, derived from CGSharedBus class, models multiple Unix machines connected together via Ethernet. We use socket mechanism for interprocessor communication. Since the communication overhead is huge, we do not gain any speed up for small examples. Nonetheless, we can test and verify the procedure of multiprocessor code generation.

This class has five private states as follows.

```
IntState doCompile;
IntState doRun;
```

If these states are set YES, we compile and run the generated code.

```
StringState machineNames;
StringState nameSuffix;
```

We list the machine names separated by commas. If all machines names listed have the same suffix, we separate that suffix in the second state. For example, if `machineName` is "ohm" and `nameSuffix` is ".berkeley.edu", we mean machine named "ohm.berkeley.edu".

```
IntState portNumber;
```

To make socket connections, we assign port numbers that are available. For now, we set the starting port number with this state. We will increase this number by one every time we add a new connection. Therefore, it should be confirmed that these assigned port numbers should be available. If the Ptolemy program is assigned a port number in the future, then we will be able to let the system choose the available port number for each connection.

With the given list of machine names, we prepare a data structure called `MachineInfo` that pairs the machine name and internet address.

```
class MachineInfo {
      friend class CGCMultiTarget;
      const char* inetAdddrr;// internet address
      const char* nm;   // machine name
public:
      MachineInfo: inetAddr(0), nm(0) { }
}
```

This class has a constructor with three argument like its base classes. The destructor deallocates `MachineInfo` arrays if allocated. It has `makeNew` method and `isA` method redefined.

18.7.1 CGCMultiTarget protected members

```
void setup();
```

If the child targets are inherited, we also inherit the machine information. Otherwise, we set up the machine information. The number of processors and the number of machines names should be equal. Then, we call `CGMultiTarget::setup` for normal setup operation . At last, we set the `hostName` state of child targets with the machine names.

```
int wormLoadCode();
```

This method do nothing if `doCompile` state is NO. Otherwise, it compiles the code for all child targets (`compileCode).` Then, it checks whether `doRun` state is YES or NO. If it is YES, we execute the code.

```
int sendWormData(PortHole& p);
int receiveWormData(PortHole& p);
int sendWormData();
int receiveWormData();
```

These method should be redefined in the future to support wormhole interface. Currently, they do same tasks with the base Target classes.

18.7.2 CGCMultiTarget public members

`MachineInfo* getMachineInfo();`
`int* getPortNumber();`

These methods return the current machine information and the next port number to be assigned.

`DataFlowStar* createSend(int` *`from`*`, int` *`to`*`, int` *`num`*`);`
`DataFlowStar* createReceive(int` *`from`*`, int` *`to`*`, int` *`num`*`);`

The above methods create CGCUnixSend and CGCUnixReceive stars for communication stars with TCP protocol.

`void pairSendReceive(DataFlowStar*` *`snd`*`, DataFlowStar*` *`rcv`*`);`

This method pairs a UnixSend star and a UnixReceive star to make a connection. We assign a port number to the connection. More important task is to generate function calls in the initialization code (`commInit` stream) of two child targets which these communication stars belong to. These functions will make a TCP connection between two child targets with the assigned port number. The UnixSend star will call `connect` function while the UnixReceive star will call `listen` function.

`void setMachineAddr(CGStar*` *`snd`*`, CGStar*` *`rcv`*`);`

This method informs the *`snd`* star about the internet address of the machine that the *`rcv`* star is scheduled on. The address is needed in `connect` function.

`void signalCopy(int` *`flag`*`);`

By giving a non-zero value as the argument, we indicate that the code will be duplicated in different set of processors so that we need to adjust the machine information of communication stars.

`void prepCode(Profile*` *`pf`*`, int` *`nP`*`, int` *`numChunk`*`);`

This method is also used to allow code replication into different set of targets.

`DataFlowStar* createCollect();`
`DataFlowStar* createSpread();`

These methods create CGCCollect and CGCSpread stars.

18.8 Status

Here are some points about the current status.

- Data Parallel star is not supported yet.
- Execution times of CGC stars are not well defined. They will vary processor to processor. We estimate them by looking at CG96 stars, or by counting the number of elementary operations. For heterogeneous multiprocessor case, we have to design a clean way of specifying these numbers.

- hSpread/Collect stars and buffer embedding are not supported in ASM domain. Since Spread/Collect stars are not supported, all ASM multiprocessor targets should set the `oneStarOneProc` state TRUE.

(4) The scheduling option, `adjustSchedule` is not implemented yet since the current graphical editor does not support "cont" function.

(5) Overlapped communication is not supported since we haven't had any machine of that sort.

18.9 References

[1] G.C.Sih and E.A.Lee, "Dynamic-level scheduling for heterogeneous processor networks," Second IEEE Symposium on Parallel and Distributed Processing, pp. 42-49, 1990

[2] G.C.Sih and E.A.Lee, "Declustering: A New Multiprocessor Scheduling Technique," IEEE Transactions on Parallel and Distributed Systems, 1992.

[3] S. Ha, Compile-time Scheduling of Dataflow Program Graphs with Dynamic Constructs, Ph.D. dissertation, U.C.Berkeley, 1992.

[4] J.L.Pino, S.Ha, E.A.Lee, J.T.Buck, "Software Synthesis for DSP Using Ptolemy," invited paper, Journal of VLSI Signal Processing, 1993.

[5] W.S.Wang, et al, "Assignment of Chain-Structured Tasks onto Chain-structured Distributed Systems," source unknown.

Index