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 func-

tion 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.