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.