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 com-

piled 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 compat-

ible 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.