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 YYYFromUniveral 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`.