Chapter 17. Creating New Domains

Authors: *Mike Chen*
Christopher Hylands
Thomas M. Parks

Other Contributors: *Wan-Teh Chang*
Michael C. Williamson

17.1 Introduction

One of Ptolemy's strengths is the ability to combine heterogeneous models of computation into one system. In Ptolemy, a model of computation corresponds to a `Domain`. The code for each `Domain` interacts with the Ptolemy kernel. This overview describes the general structure of the various classes that are used by a `Domain` in its interaction with the kernel. The Ptolemy *User's Manual* has a more complete overview of this information.

A functional block, such as an adder or an FFT, is called a `Star` in Ptolemy terminology, (see "Writing Stars for Simulation" on page 2-1 for more information). A collection of connected `Stars` form a `Galaxy` (see Chapter 2 of the *User's Manual* for more information). Ptolemy supports graphical hierarchy so that an entire `Galaxy` can be formed and used as a single function block icon. The `Galaxy` can then be connected to other `Stars` or `Galaxies` to create another `Galaxy`. Usually, all the `Stars` of a `Galaxy` are from the same `Domain` but it is possible to connect `Stars` of one domain to a `Galaxy` of another domain using a `WormHole`.

A `Universe` is a complete executable system. A `Universe` can be either a single `Galaxy` or a collection of disconnected Galaxies. To run a `Universe`, each `Galaxy` also needs a `Target`. In simulation domains, a `Target` is essentially a collection of methods to compute a schedule and run the various `Stars` of a `Galaxy`. Some `Domains` have more than one possible scheduling algorithm available and the `Target` is used to select the desired scheduler. In code generation domains, a `Target` also computes a schedule and runs the individual `Stars`, but each `Star` only generates code to be executed later. Code generation `Targets` also handle compiling, loading, and running the generated code on the target architecture.

At a lower level are the connections between `Blocks`. A `Block` is a `Star` or `Galaxy`. Each `Block` has a number of input and output terminals which are attached to a `Block` through its `PortHoles`. A special `PortHole`, called a `MultiPortHole`, is used to make multiple connections but with only one terminal. Two `Blocks` are not directly connected through their `PortHoles`. Rather, their `PortHoles` are connected to an intermediary object called a `Geodesic`. In simulation domains, data is passed between `PortHoles` (through the `Geodesic`) using container objects called `Particles`. Ptolemy uses a system where `Particles` are used and recycled instead of created and deleted when needed. `Particles` are obtained from a production and storage class called a `Plasma`, which creates new `Particles` if there are no old ones to reuse. `Particles` that have completed their task are returned to the

Plasma, which may reissue them at a later request. Graphically, the Star to Star connection is depicted below:

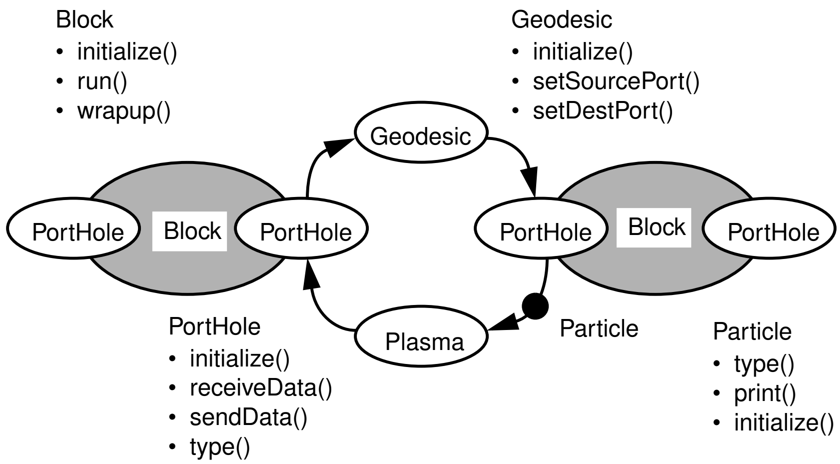

FIGURE 17-1: Block objects in Ptolemy can send and receive data encapsulated in Particles through Portholes. Buffering and transport is handled by the Geodesic and garbage collection by the Plasma. Some methods are shown.

The classes defined above provide most of the functionality necessary for a working domain. One additional class needed by all domains is a Scheduler to compute the order of execution of the Stars in the Galaxy.

Therefore, creating a new Ptolemy simulation domain will typically involve writing new classes for Stars, PortHoles, WormHoles, Targets, and Schedulers.

Creating a new domain is a fairly involved process, and not to be done lightly. The first thing that many users want to do when they see Ptolemy is create a new domain. However, it is often the case that the functionality they need is already in either the SDF or DE domains, or they can merely add a Target or Scheduler rather than an entire domain.

17.2 A closer look at the various classes

A simulation Domain can use the various classes mentioned above as they exist in the Ptolemy kernel or it can redefine them as needed. For example, in the SDF domain, the classes SDFStar, SDFPortHole, SDFScheduler, SDFDomain, SDFTarget, and SDFWormhole have all been defined. Most of those classes inherit much of their functionality from the corresponding kernel classes but the Domain creator is free to make major changes as well. The kernel Geodesic, Plasma, and Particle classes are used without modification, but other domains such as the CG domain have derived a subclass from Geodesic. The Domain creator needs to decide whether or not existing Ptolemy classes can be used without change, therefore it is a good idea to understand what functionality the kernel classes provide.

The following is a brief description of the various classes that either need to be defined or are used by a Domain. Note that we only provide a functional description of some of the major methods of each class and not a complete description of all methods.

17.2.1 Target

A `Target` is an object that manages the execution of the `Stars` in a `Domain`.

Major methods:

`run()`	Called to execute a schedule.
`wrapup()`	Called at the end of an execution to clean up.
`setup()`	Called by `initialize()` (which is inherited from the `Block` class, which is a common base class for many of Ptolemy's classes). Sets each `Star` to point to this `Target` and sets up the `Scheduler`.

Major objects contained are:

`gal`	A pointer to the `Galaxy` being executed.
`sched`	A pointer to the Scheduler that is being used.

For further information about `Targets`, see some of the existing domains.

17.2.2 Domain

Declares the type of various components of the `Domain`, like which type of `WormHole`, `PortHole`, `Star`, etc. is used by the `Domain`.

Major methods:

`newWorm()`	Create a `WormHole` of the appropriate type for this `Domain`.
`newFrom()`	Create an `EventHorizon` (an object that is used to interface to other `Domains`, used with `WormHoles`) that translates data from a Universal format to a `Domain` specific one.
`newTo()`	Create an `EventHorizon` that translates data from a `Domain` specific format to a Universal one.
`newNode()`	Returns a `Geodesic` of the appropriate type for this `Domain`.

17.2.3 Star

A `Star` is an object derived from class `Block` that implements an atomic function.

Major methods:

`run()`	What to do to run the star.

For example, the `DataFlowStar` class (a parent class to many of the dataflow domain stars such as `SDFStar` and `DDFStar`) defines this function to make each input `PortHole` obtain `Particles` from the `Geodesic`, execute the `go()` method of each `Star`, and then have each output `PortHole` put its `Particles` into the `Geodesic`.

17.2.4 PortHole

`PortHoles` are data members of `Stars` and are where streams of `Particles` enter or leave the `Stars`. Each `PortHole` always handles `Particles` of one type, so two connected `PortHoles` need to decide which data type they will use if they are not the same. There is a base class called `GenericPort` which provides some basic methods that derived classes should redefine as well as some data members commonly needed by all `PortHole` types.

Major methods:

`isItInput()`	Return `TRUE` if the `PortHole` class is an input type.
`isItOutput()`	Return `TRUE` if the `PortHole` class is an output type.
`isItMulti()`	Return `TRUE` if the `PortHole` class is a `MultiPorthole`.
`connect()`	Connect this `PortHole` to a `Geodesic` (create one if needed) and tell that `Geodesic` to connect itself to both this `PortHole` and the destination `PortHole`. Also provides the number of delays on this connection.
`initialize()`	Initialize the `PortHole`. In the case of output `PortHoles`, this function will usually initialize the connected `Geodesic` as well. Resolve the type of `Particles` with the `PortHole` it is connected to.
`receiveData()`	What to do to receive data from the `Geodesic`.
`sendData()`	What to do to send data to the `Geodesic`.
`putParticle()`	Put a particle from the buffer into the `Geodesic`.
`getParticle()`	Get a particle from the `Geodesic` and put it into the buffer.
`numXfer()`	Returns `numberTokens`, the number of `Particles` transferred per execution.
`numTokens()`	Returns the number of `Particles` inside the `Geodesic`.
`numInitDelays()`	Returns the number of initial delay on the `Geodesic`.
`geo()`	Returns a pointer to the `Geodesic` this `PortHole` is connected to.
`setDelay()`	Set the delay on the `Geodesic`.

Major data members:

`myType`	Data type of particles in this porthole.
`myGeodesic`	The `Geodesic` that this `PortHole` is connected to
`myPlasma`	A pointer to the `Plasma` used to request new `Particles`.
`myBuffer`	Usually a `CircularBuffer` used to store incoming or outgoing `Particles`.
`farSidePort`	The `PortHole` that we are connected to.
`bufferSize`	The size of the `Buffer`.
`numberTokens`	The number of `Particles` consumed or generated each time we access the `Geodesic`.

Note that `PortHoles` are generally separated into input `PortHoles` and output

`PortHoles`. They aren't designed to handle bidirectional traffic.

17.2.5 Geodesic

Models a FIFO buffer (usually) between two `PortHoles`. Major methods:

`setSourcePort()` Set the source `PortHole` and the delay on this connection. A delay is usually implemented as an initial `Particle` in the `Geodesic`'s buffer, but this can be changed depending on the desired functionality.

`setDestPort()` Set the destination `PortHole`.

`disconnect()` Disconnect from the given `PortHole`.

`setDelay()` Set the number of delays on this connection.

`initialize()` Initialize the buffer in this `Geodesic`. This means either clear it or insert the number of initial `Particles` needed to match the number of delays on this connection (these `Particles` are taken from the source `PortHoles`'s Plasma).

`put()` Put a `Particle` into the buffer

`get()` Get a `Particle` from the buffer. `incCount()` and `decCount()` are used by a `Scheduler` to simulate an execution.

`numInit()` Return the number of initial particles.

Major data members:

`originatingPort` A pointer to the source `PortHole`.

`destinationPort` A pointer to the destination `PortHole`.

`pstack` The buffer, implemented as a `ParticleStack`.

`sz` The number of `Particles` in the buffer.

`numInitialParticles`
The number of initial delays.

17.2.6 Plasma

There are container object for unused `Particles`. There is one global instance of a Plasma for each type of `Particle` defined in the kernel. This class is usually only used by the `Domains` and not changed by the authors of new `Domains`.

Major methods:

`put()` Return an unused `Particle` to the `Plasma`.

`get()` Get an unused `Particle` (or create one if needed).

17.2.7 Particle

The various `Particle` types supported by Ptolemy. Currently, the types are `Float`, Int, Complex, Fix, and Message. The Message Particle is used to carry Messages (inside Envelopes) which can be almost anything. For example, the Matrix class is transferred using Message Particles. These classes are also only used as-is by the Domains and not redefined for new domains.

17.2.8 Scheduler

Sets up the execution by determining the order in which each Star of the Galaxy will fire. Execution is performed using two main methods -- setup() and run(). Schedulers can be timed or untimed, depending on the Domain's model of execution. This class will usually be different for each domain, although some domains reuse the Scheduler of another domain, if the Scheduler is appropriate for the new domain's model of computation.

Major methods:

setup()	Checks the Stars in the Galaxy, initializes them, and creates a schedule.
run()	Run the schedule computed in setup()

Major data members

myGalaxy	The pointer to the Galaxy that the Scheduler is working on.
myTarget	The pointer to the Target which is controlling the execution.

17.3 What happens when a Universe is run

Now that you have some idea of what classes exist in the Ptolemy kernel, this section will try to explain flow of control when a Universe is run. By knowing this, you will get an idea of what additions or changes might be needed to get the functionality you desire and how the code of your new domain will fit in.

First off, a little more about the basics of Ptolemy classes. Almost every object class in Ptolemy is derived from the NamedObj class. This class simply provides support for a Name field, a longer Description field, and a pointer to a Parent Block. Also, the method initialize() is declared here to be purely virtual, so every object should have some kind of initialization function.

The Block class is derived from NamedObj and is the main base class for most actors in Ptolemy. It has I/O constructs like PortHoles and MultiPortHoles, state/parameter constructs like State, and defines execution methods such as setup(), run() and wrapup(). The Block also provides a virtual function to access an associated Scheduler.

A simulation universe is generally of type DataFlowStar. When a universe is run, the flow of control is as follows, using the SDF domain as an example:

```
PTcl::dispatcher()
    PTcl::run()
        PTcl::computeSchedule()
            Runnable::initTarget()
                Block::initialize()
                    SDFTarget::setup()
                        Target::setup()
                            SDFScheduler::setup()
```

Notice at this point that we have called two domain-specific methods, namely `SDFTarget::setup()` and `SDFScheduler::setup()`. The `Target` can have a choice of more than one `Scheduler` and in this case it called the default `SDFScheduler`. We continue here with a more detailed description of a very important function:

```
SDFScheduler::setup()
checkConnectivity()     // Checks that the galaxy is
                        //   properly connected.
  prepareGalaxy()       // Initializes the portHoles of each star and
                        //   the geodesics that connect them.
  checkStars()          // Verifies that the type of the Stars are
                        //   compatible with this Scheduler.
  repetitions()         // Solves the balance equations for the
                        //   system and calculates how many times
                        //   each star should be fired for
                        //   one iteration (specific to dataflow).
  computeSchedule()     // Compute the actual schedule
  adjustSampleRates()   // Set the number of tokens transferred
                        //   between EventHorizons if this schedule
                        //   is for a WormHole.
```

The order of various operations can be different for each scheduler. For example, a new domain may require that the `PortHoles` be initialized after the repetitions were calculated but before the schedule was computed. The domain writer may wish to define a new function `prepareForScheduling()` that would call the `setup()` function of each `Star` without initializing the `Star`'s `PortHoles`.

Expanding `prepareGalaxy()` in more detail:

```
SDFScheduler:: prepareGalaxy()
 galaxy()->initialize()       // Initialize the galaxy.
  InterpGalaxy::initialize()  // Causes the initialization of delays
                              // and the setup of bus widths.
   Galaxy::initSubblocks()    // Calls initialize() of each star.
    DataFlowStar::initialize()// This is a general initialize.
                              // function for data flow stars.
                              //   Your own Star class might
                              //   redefine it. Sets the number
                              //   of input Ports and clears
                              //   some parameters.
     Block::initialize()      // Initializes the PortHoles and States
                              //   of the Block/Star. Calls the user
                              //   defined setup() function of each
                              //   star after the portholes and
                              //   geodesics have been initialized.
      PortHole::initialize()  // General PortHole initialization;
                              //   again you can redefine it for a
                              //   domain specific PortHole.
                              //   Resolves the type of Particles
                              //   to be sent. Allocates a
                              //   buffer and a Plasma. Request
                              //   empty Particles from the Plasma
                              //   to initialize the buffer.
       Geodesic::initialize() // General Geodesic initialization,
```

```
                                   //   called by output PortHole only.
                                   //   Clears the buffer and adds any
                                   //   initial Particles for delays.
```

After the schedule is set up and all the actors in the `Universe` have been initialized, the flow of control is as follows:

```
PTcl::run()
 PTcl::computeSchedule()      // Described above.
  PTcl::cont()
   universe->setStopeTime()   // Used to set the number of
                              //   iterations to be run.
    universe->run()
     InterpUniverse::run()
      Runnable::run()
       target->run()
        sched->run()
         SDFScheduler::run()  // The domain specific Scheduler's
                              //   run() function.
```

Let's look at what a typical scheduler does when it runs a star.

```
SDFScheduler::run()           // Checks if there has been an error
                              //   in the last iteration. Calls
                              //   runOnce() for each iteration.
 runOnce()                    //   Goes through each Star on the
                              //   schedule (which is a list of Stars
                              //   computed by setup() ) and calls
                              //   star->run().
  star->run()
   DataFlowStar::run()        // The SDF domain uses the general
                              //   DataFlowStar
                              //   run() function. A new Domain
                              //   might want to redefine this.
    ..Ports->receiveData()    // Calls receiveData() for each of
                              //   the PortHoles for this Star.
                              //   Output PortHoles would do nothing
                              //   in this case but input PortHoles
                              //   would get Particles from the
                              //   Geodesic.
    Star::run()
     SimControl::doPreActions()// Execute pre-actions for a star.
     go()                     // Call the Star specific go() function
                              //   that will process the input data
                              //   and generate data to be put in the
                              //   output PortHoles.
     SimControl::doPostActions() // Execute post-actions for a star
     ..Ports->sendData()      // Calls sendData() for each of the
                              //   PortHoles for this Star.
                              //   Input PortHoles would do nothing
                              //   in this case but output PortHoles
                              //   would put their Particles into
                              //   the Geodesic and refill their
                              //   buffers with empty Particles
                              //   from the Plasma.
```

17.4 Recipe for writing your own domain

This section describes some of the template files we have made so that you don't have to start coding from scratch. We also discuss which classes and methods of those classes that a new domain must define.

17.4.1 Introduction

The first thing to do is to think through what you want this domain to do. You should have some idea of how the your `Stars` will exchange data and what kind of `Scheduler` is needed. You should also understand the existing Ptolemy domains so that you can decide whether your domain can reuse some of the code that already exists. Also, read Chapter 1 so you understand the general classes in the Ptolemy kernel and how the domain methods interact.

17.4.2 Creating the files

The `mkdom` script at `$PTOLEMY/bin/mkdom` can be used to generate template files for a new domain. `mkdom` takes one argument, the name of the domain, which case insensitive, `mkdom` converts the what ever you pass to it as a domain name to upper and lower case internally. Here, we assume that you have set up a parallel development tree, as documented in chapter 1, or you are working in the directory tree where Ptolemy was untar'd.

1. To use `mkdom`, create a directory with the name of your domain in the `src/domains` directory. In this example, we are creating a domain called `yyy`:

```
mkdir $PTOLEMY/src/domains/yyy
```

2. cd to that directory and then run `mkdom`:

```
cd $PTOLEMY/src/domains/yyy
$PTOLEMY/bin/mkdom yyy
```

17.4.3 Required classes and methods for a new domain

`mkdom` will create copies of key files in `$PTOLEMY/src/domains/yyy/kernel` and a `Nop` star in `$PTOLEMY/src/domains/yyy/stars`. The template files have various comments about which methods you need to redefine. The template files also define many function for you automatically. If you aren't clear as to how to define the methods in each class, it is best to try look at the existing Ptolemy domains as examples.

`*YYY*Domain.cc` This file will be setup for you automatically so that you shouldn't need to modify much. The various methods here return `WormHoles` and `EventHorizons` which should be defined in `*YYY*Wormhole`. A node is usually a type of `Geodesic` that allows multiple connections, such as `AutoForkNode`. You can define your own `*YYY*Geodesic` or simply use the kernel's `AutoForkNode` if that is suitable (this is what SDF does).

`*YYY*Wormhole.{h,cc}`
Various methods to interface your new domain with others must be defined if you wish to use your domain with other domains.

However, if you don't need to mix domains, then you may skip these files. Wormholes translate different notions of time or concurrency. Since some domains are timed (like DE) and others are not (like SDF), you must be able to convert from one to another.

`YYYGeodesic.{h,cc}`
: Currently we set the `Geodesic` to be the kernel's `AutoForkNode`. If the kernel's `Geodesic` class offers all the functionality you need, then this doesn't need to be changed. Otherwise try looking at some of the pre-existing domains for examples.

`YYYPortHole.{h,cc}`
: Define input `PortHoles` and output `PortHoles`, as well as `MultiPortHoles`, specific to your domain. The only required methods are generated for you, but you'll likely want to define many more support methods. Look at the kernel `PortHole`, `DFPortHole`, and `SDFPortHole` for examples.

`YYYStar.{h,cc}`
: `Domain`-specific class definition. Again, all the required methods have been defined but you'll want to add much more. Refer to `Star`, `DataFlowStar`, and `SDFStar` as examples.

`YYYScheduler.{h,cc}`
: This is where much of the action goes. You'll need to define the function `setup()`, `run()`, and `setStopTime()`.

17.4.4 Building an object directory tree

Ptolemy can support multiple machine architectures from one source tree, the object files from each architecture go into `$PTOLEMY/obj.$PTARCH` directories. Currently, there are two ways to build the `$PTOLEMY/obj.$PTARCH` directory tree: `MAKEARCH` and `mkPtolemyTree`. To build object files for your new domain in `$PTOLEMY/obj.$PTARCH`, you will have to set up either or both of these ways. Typically, you first use `MAKEARCH` because it can operate on an existing Ptolemy tree, and once everything works, then you and other users run `mkPtolemyTree` to setup parallel development trees on the new domain.

MAKEARCH

`$PTOLEMY/MAKEARCH` is a `/bin/csh` script that creates or updates the object tree in an already existing Ptolemy tree. To add a domain to `MAKEARCH`, edit the file and look for a similar domain, and add appropriately. A little trial and error may be necessary, but the basic idea is simple: `MAKEARCH` traverses directories and creates subdirectories as it sees fit. Note that if `MAKEARCH` is under version control, you may need to do `chmod a+x MAKEARCH` when you check it back out, or it won't be executable.

Continuing with our example:

3. Edit `MAKEARCH` and add your domain yyy to the list of experimental domains:

```
set EXPDOMAINS=(cg56 cgc vhdlb vhdl mdsdf hof ipus yyy)
```

This will cause a `stars` and `kernel` directory to be created in `$PTOLEMY/obj.$PTARCH/domains/yyy` when `MAKEARCH` is run.

4. Run `MAKEARCH`:

```
cd $PTOLEMY; csh -f MAKEARCH
```

If you get a message like:

```
cxh@watson 181% csh -f MAKEARCH
making directory /users/ptolemy/obj.sol2/domains/yyy
mkdir: Failed to make directory "yyy"; Permission denied
yyy: No such file or directory
```

The you may need to remove your `obj.$PTARCH` tree, as `MAKEARCH` has probably traversed down a parallel tree created by `mkPtolemyTree` and come up in a directory that you do not own.

mkPtolemyTree

`$PTOLEMY/bin/mkPtolemyTree` is a `tclsh` script that creates a new parallel Ptolemy tree. Note that `mkPtolemyTree` cannot be run in an already existing Ptolemy development tree. The file `$PTOLEMY/mk/stars.mk` controls what directories `mkPtolemyTree` creates, you need not actually edit the `mkPtolemyTree` script. To create `pigiRpc` binaries with your new domain in it, you will need to modify `stars.mk`, so adding support for `mkPtolemyTree` is fairly trivial.

$PTOLEMY/mk/stars.mk

Follow the style for domain addition that you see in this file for the other domains. A few things to keep in mind:

- You should list the new domain before any other domain library that the new domain depends on.
- You should make sure to define the make variables to pull in other domain libraries as necessary. You may need `MDSDF=1` definition for example.
- `mkPtolemyTree` uses the `CUSTOM_DIRS` makefile variable to determine what directories to create, so be sure to add your directories here.

Continuing with our example of adding the yyy domain:

5. Edit `$PTOLEMY/mk/stars.mk` and add your entry:

```
YYYDIR = $(CROOT)/src/domains/cg56
ifdef YYY
      CUSTOM_DIRS += $(YYYDIR)/kernel $(YYYDIR)/stars
      # Have to create this eventually
      PALETTES += PTOLEMY/src/domains/yyy/icons/main.pal
      STARS += $(LIBDIR)/yyystars.o
      LIBS += -lyyystars -lyyy
      LIBFILES += $(LIBDIR)/libyyystars.$(LIBSUFFIX) \
            $(LIBDIR)/libyyy.$(LIBSUFFIX)
endif
```

$PTOLEMY/mk/ptbin.mk

In `$PTOLEMY/mk/ptbin.mk`, add your domain to the `FULL` definition. This causes your domain to be built in whenever a full `pigiRpc` binary is created.

Building a pigiRpc

6. To build a `pigiRpc` with your domain, first build and install your domain's kernel and star libraries:

```
cd $PTOLEMY/obj.$PTARCH/domains/yyy
make depend
make install
```

   If your domain depends on other domains, you will have to build in those directories as well. You may find it easier to do `cd $PTOLEMY; make install`, though this could take 3 hours. An alternative would be to create a parallel directory tree using `mkPtolemyTree`.

7. If you have not recompiled from scratch, or run `mkPtolemyTree`, you may also need to do:

```
cd $PTOLEMY/obj.$PTARCH/pigilib; make ptkRegisterCmds.o
```

8. Then build your `pigiRpc`. You can either build a full `pigiRpc` with all of the domains, or you can create a `override.mk` in `$PTOLEMY/obj.$PTARCH/pigiRpc` which will pull in only the domains you want.

   `$PTOLEMY/obj.$PTARCH/pigiRpc/override.mk` could contain:

```
YYY=1
DEFAULT_DOMAIN=YYY
USERFLAGS=
VERSION_DESC="YYY Domain Only"
```

   To build your binary, do:

```
cd $PTOLEMY/obj.$PTARCH/pigiRpc; make
```

   If you don't have all the libraries built, you may get an error message:

```
make: *** No rule to make target `../../lib.sol2/libcg56dspstars.so',
needed by `pigiRpc'. Stop.
```

   The workaround is to do:

```
cd $PTOLEMY/obj.$PTARCH/pigiRpc; make PIGI=pigiRpc
```

9. See "Creating a pigiRpc that includes your own stars" on page 1-7 for details on how to use your new pigiRpc binary.

10. To verify that your new domain has been installed, start `pigi` with the `-console` option:

```
cd $PTOLEMY; pigi -rpc $PTOLEMY/obj.$PTARCH/pigiRpc/pigiRpc -console
```

and then type:

```
domains
```

into the console window prompt. Below is the sample output for the yyy example domain:

```
pigi> domains
YYY
pigi> knownlist
Nop
pigi>
```