Chapter 1. Extending Ptolemy — Introduction

Authors: *Christopher. Hylands*
Edward. A. Lee
Thomas. M. Parks
José Luis Pino

1.1 Introduction

Ptolemy is extensible in the following ways:

- New galaxies can be defined. We do not view this as a programming task, so it is explained in the *User's Manual* rather than in this *Programmer's Manual*.
- Customized simulation builders and controllers can be created using the `ptcl` interpreted command language. This language is also covered in the *User's Manual*.
- New functional blocks (stars) can be added to any of the Ptolemy domains. These blocks can be dynamically linked with either `ptcl` or `pigi`.
- New code generation blocks can be added to existing synthesis domains.
- Stars with customized user interfaces and displays can be created using Tcl/Tk.
- New simulation and design-flow managers (called targets) can be created.
- New domains with new models of computation can be created.

This volume explains how to accomplish most of the above. The *Kernel Manual*, volume 3 of *The Almagest*, supplements this volume with a detailed listing of all of the classes in the Ptolemy kernel and in the code generation kernel. The sophisticated user, however, who is extending the system in nontrivial ways, will wish to refer to the source code as the ultimate, most complete documentation.

In this volume, we assume familiarity with the terminology and use of Ptolemy. Refer to the *User's Manual*, and particularly to the glossary contained therein for assistance. We also assume you are familiar with the overall organization of the Ptolemy software, as described in *User's Manual*.

1.2 File Organization

Ptolemy is distributed with source code. The complete distribution even includes the compiler we use (g++, from the Free Software Foundation), Tcl/Tk, and `vem`, programs that were developed quite independently, but upon which Ptolemy relies. The distribution also includes a large number of demonstrations. Perusing the demonstrations can be an excellent way to get familiar with the system. Perusing the source code is by far the best way to understand the system. At a minimum, anyone wishing to write new stars should read the source code for a few of the built-in stars.

1.2.1 Ptolemy environment variables

The root of the Ptolemy tree is often installed in the home directory of a fictitious user called `ptolemy`. If the installation follows this model at your site, you can find the Ptolemy code with the following command:

```
cd ~ptolemy
```

If your installation does not have a user named `ptolemy`, then you must find out where your system administrator has installed the system, and set an environment variable called `PTOLEMY` to point to this directory. For instance, if your system administrator installed Ptolemy in `/users/ptolemy`, then you should issue the following command:

```
setenv PTOLEMY /users/ptolemy
```

`$PTARCH` is an environment variable representing the architecture on which you are running, and has one (or more) of the following values: `sun4`, `sol2`, or `hppa`, for Sun (under Sun O/S), Sun (under Solaris 2.X), and HP machines respectively. There are a few other possible values for the `PTARCH` variable as well. There might be variations like `sol2.cfront` or `hppa.cfront` to store an object tree created by the Cfront C++ compiler or some other non-g++ compiler.The script `$PTOLEMY/bin/ptarch` will return the architecture of the machine on which it is run. For example, if you were on a machine running SunOS4.1.3, you would type:

```
setenv PTARCH sun4
```

You can use the following fragment in your `.cshrc` file to set `$PTOLEMY` and `$PTARCH`. The `$PTOLEMY/.cshrc` file contains the fragment below and many other csh setup commands you may find useful.

```
setenv PTOLEMY /users/ptolemy
if (! $?$PTARCH) setenv $PTARCH ` $PTOLEMY/bin/ptarch`
set path = ( $PTOLEMY/bin $PTOLEMY/bin.$PTARCH $path )
```

Note that if you are using a prebuilt Gnu compiler that you obtained from the Ptolemy project, you must either place the Ptolemy distribution at `/users/ptolemy`, or you must set certain environment variables so that the Gnu compiler can find the necessary pieces of itself. Appendix A, Installation and Troubleshooting of the Ptolemy *User's Manual* discusses these variables in detail. The variables change with different releases of the compilers, so we do not document them here. The User's Manual also documents other useful environment variables, such as `LD_LIBRARY_PATH`.

For every directory under the `src` tree (see figure 1-2) that contains source code that is compiled, there is a corresponding directory under the `obj.$PTARCH` tree. Many developers find it convenient to set the following aliases:

```
alias srcdir ‘cd `pwd | sed “s?/obj.$PTARCH/?/src/?”`’
alias objdir ‘cd `pwd | sed “s?/src/?/obj.$PTARCH/?”`’
```

For your convenience, these can be found in the file `$PTOLEMY/.alias`. They make it easy to move between the source directory and the corresponding object directory. For example, if you are running on a Sun machine running Solaris 2.4,

```
% cd $PTOLEMY/src/kernel
% pwd
/users/ptolemy/src/kernel
% objdir
% pwd
/users/ptolemy/obj.sol2/kernel
% srcdir
% pwd
/users/ptolemy/src/kernel
%
```

1.2.2 Directory Structure

The documentation (usually) refers to the root of the Ptolemy directory tree as `$PTOLEMY`, but occasional slips will refer to `~ptolemy`. Below this root, you can find the directories indicated in figure 1-1.

The `src` directory is key to much of what this volume deals with. Its structure is shown in figure 1-2. Within the `src` directory, the `kernel` directory is most important. It contains all the classes that define what Ptolemy is. Second most important is the `domains` directory. Its structure is shown in figure 1-3. This directory contains one subdirectory that defines each of the domains distributed with Ptolemy. Each domain directory contains at least the subdirectories shown in figure 1-4. If you are going to write stars for the SDF domain, for example, then you would be well advised to look at a few examples contained in the directory `$PTOLEMY/src/domains/sdf/stars`.

The directory `$PTOLEMY/mk` contains master makefiles that are included by other makefiles (The makefile `include` directive does this for us). `$PTOLEMY/mk/config-$PTARCH.mk` refers to the makefile for the architecture `$PTARCH`. For instance, `$PTOLEMY/mk/config-sun4.mk` is the makefile that contains the sun4 specific details.

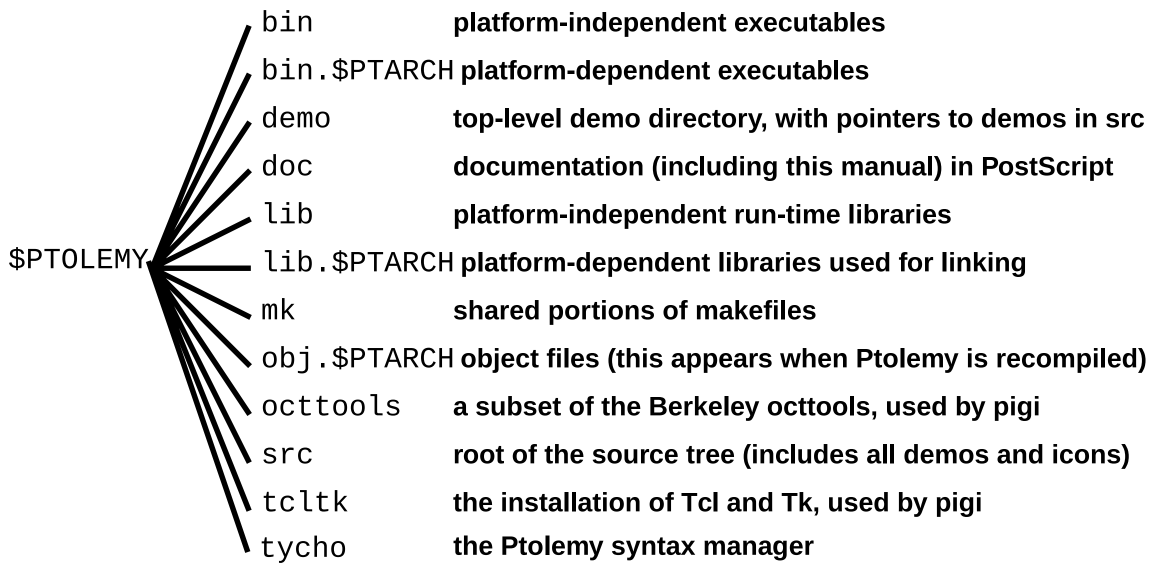

$PTOLEMY
bin
platform-independent executables
bin.$PTARCH
platform-dependent executables
demo
top-level demo directory, with pointers to demos in src
doc
documentation (including this manual) in PostScript
lib
platform-independent run-time libraries
lib.$PTARCH
platform-dependent libraries used for linking
mk
shared portions of makefiles
obj.$PTARCH
object files (this appears when Ptolemy is recompiled)
octtools
a subset of the Berkeley octtools, used by pigi
src
root of the source tree (includes all demos and icons)
tcltk
the installation of Tcl and Tk, used by pigi
tycho
the Ptolemy syntax manager

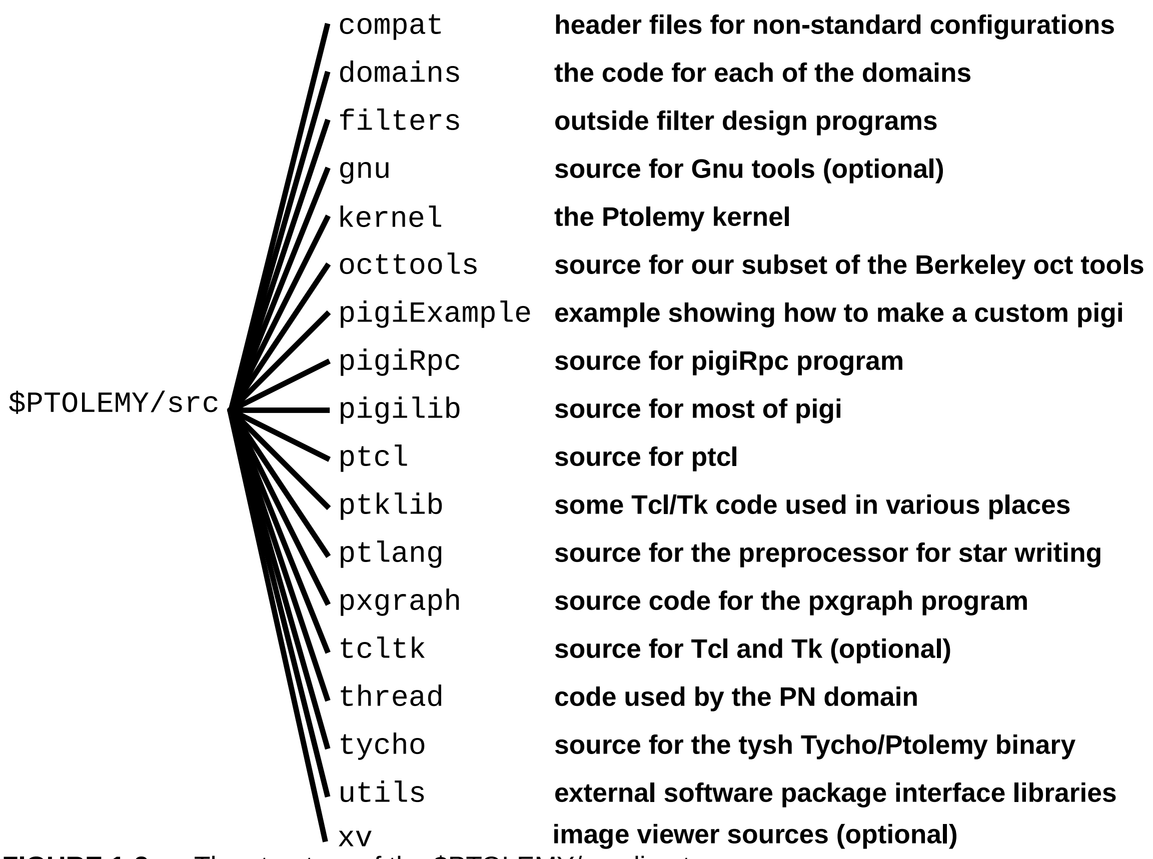
FIGURE 1-1: Structure of the home directory of the Ptolemy installation ($PTOLEMY).

When you cd to `$PTOLEMY` and type `make`, `$PTOLEMY/makefile` contains a rule that checks to see if the directory `$PTOLEMY/obj.$PTARCH` exists. If this directory does not exist, then make runs the command `csh -f MAKEARCH`, where `MAKEARCH` is a C shell script at `$PTOLEMY/MAKEARCH`. `MAKEARCH` will create the necessary subdirectories under `$PTOLEMY/obj.$PTARCH` for `$PTARCH` if they do not exist.

We split up the sources and the object files into separate directories in part to make it easier to support multiple architectures from one source tree. The directory `$PTOLEMY/obj.$PTARCH` contains the platform-dependent object files for a particular architecture. The platform-dependent binaries are installed into `$PTOLEMY/bin.$PTARCH`, and the libraries go into `$PTOLEMY/lib.$PTARCH`. Octtools, Tcl/Tk, and Gnu tools have their own set of architecture-dependent directories.

The makefiles are all designed to be run from the `obj.$PTARCH` tree so that object files from different platforms are kept separate (when you run `make` in the `$PTOLEMY` top level, the appropriate `obj.$PTARCH` tree is selected for you automatically).

We are able to have separate object and source directories by using the `make` program's `VPATH` facility. Briefly, `VPATH` is a way of telling `make` to look in another directory for a file if that file is not present in the current directory. For more information, see the Gnu `make` documentation, in Gnu Info format files in `$PTOLEMY/gnu/common/info/make-*`.

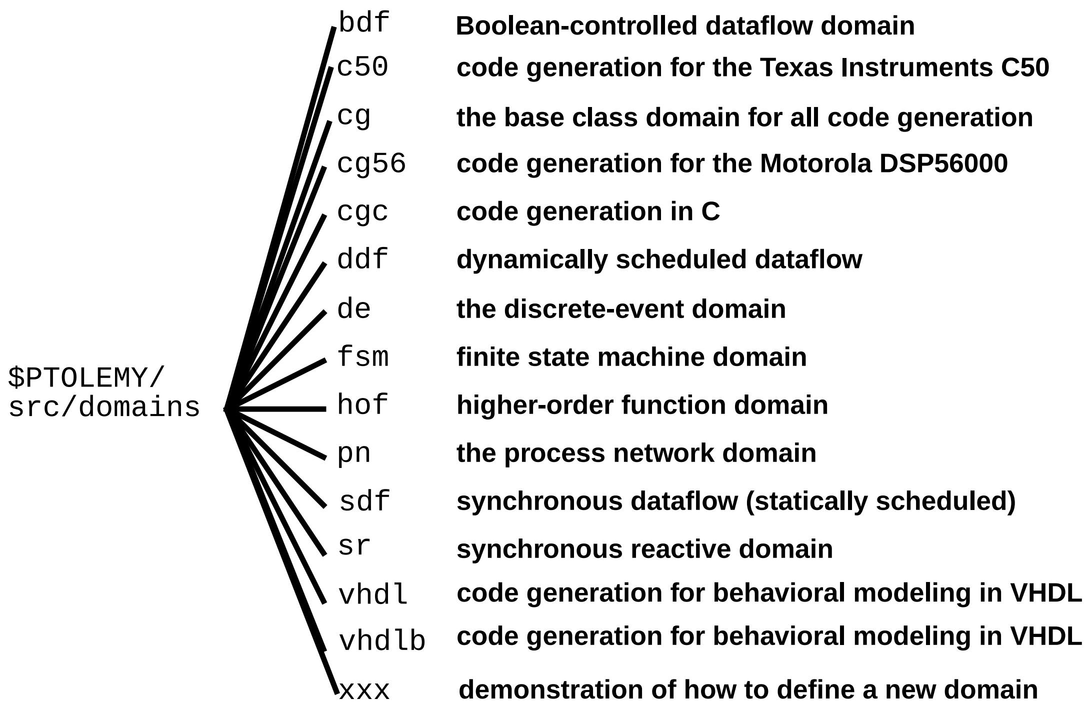


compat	**header files for non-standard configurations**
domains	**the code for each of the domains**
filters	**outside filter design programs**
gnu	**source for Gnu tools (optional)**
kernel	**the Ptolemy kernel**
octtools	**source for our subset of the Berkeley oct tools**
pigiExample	**example showing how to make a custom pigi**
pigiRpc	**source for pigiRpc program**
pigilib	**source for most of pigi**
ptcl	**source for ptcl**
ptklib	**some Tcl/Tk code used in various places**
ptlang	**source for the preprocessor for star writing**
pxgraph	**source code for the pxgraph program**
tcltk	**source for Tcl and Tk (optional)**
thread	**code used by the PN domain**
tycho	**source for the tysh Tycho/Ptolemy binary**
utils	**external software package interface libraries**
xv	**image viewer sources (optional)**

FIGURE 1-2: The structure of the $PTOLEMY/src directory

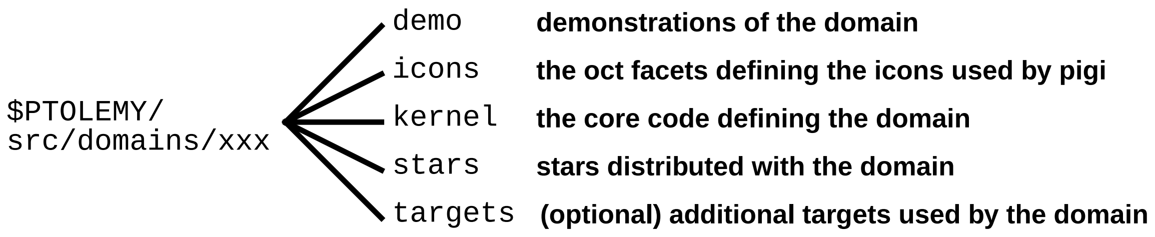
There are three primary Ptolemy binaries:

`pigiRpc`	The graphical version that uses `vem` as a front end. `pigiRpc` contains an interface to Octtools, the package that is used to store facets. When you run `pigi`, you actually run a script called `$PTOLEMY/bin/pigiEnv.csh` which calls `vem` which, in turn, starts up `pigiRpc`.
`ptcl`	A prompt version that contains most of the functionality in `pigiRpc` not including the Tk stars. `ptcl` does not contain an interface to Octtools
`tysh`	The Tycho shell version, which is similar to `pigiRpc`, except that `tysh` does not contain an interface to Octtools. Note that Tycho can be run from a basic `itkwish` binary that contains no Ptolemy functionality.



`bdf`	**Boolean-controlled dataflow domain**
`c50`	**code generation for the Texas Instruments C50**
`cg`	**the base class domain for all code generation**
`cg56`	**code generation for the Motorola DSP56000**
`cgc`	**code generation in C**
`ddf`	**dynamically scheduled dataflow**
`de`	**the discrete-event domain**
`fsm`	**finite state machine domain**
`hof`	**higher-order function domain**
`pn`	**the process network domain**
`sdf`	**synchronous dataflow (statically scheduled)**
`sr`	**synchronous reactive domain**
`vhdl`	**code generation for behavioral modeling in VHDL**
`vhdlb`	**code generation for behavioral modeling in VHDL**
`xxx`	**demonstration of how to define a new domain**

FIGURE 1-3: The structure of the $PTOLEMY/src/domains directory.



`demo`	**demonstrations of the domain**
`icons`	**the oct facets defining the icons used by pigi**
`kernel`	**the core code defining the domain**
`stars`	**stars distributed with the domain**
`targets`	**(optional) additional targets used by the domain**

FIGURE 1-4: The structure of a typical domain directory within $PTOLEMY/src/domains.

Each of the three binaries above has three different versions that contain different functionality. Below we only list the different version of `pigiRpc`, but `ptcl` and `tysh` have similar versions.

`pigiRpc`	This binary contains all of the domains, so it is the largest binary.
`pigiRpc.ptrim`	This binary contains SDF, DE, BDF, DDF and CGC domains only.
`pigiRpc.ptiny`	This binary contains SDF (no image stars) and DE domains only.

Each of the above versions can also be built as a `.debug` version that contains debugging information. The file `$PTOLEMY/mk/ptbin.mk` contains rules to build the above binaries in combination with debugging and other features. The file `$PTOLEMY/mk/stars.mk` contains rules that indicate dependencies between domains and other features.

1.3 Creating Custom Versions of pigiRpc

Ptolemy is an extensible system. Extensions can take the form of universes and galaxies, which are viewed by Ptolemy as applications, but they can also take the form of additional code linked to the Ptolemy kernel. New stars can be dynamically linked (see “Writing Stars for Simulation” on page 2-1). Other additional code has to be linked in statically. If you add many of your own stars to the system, you will want these stars to be statically linked as well, so that you do not have to wait for the dynamic linking to complete every time you execute your applications.

The Ptolemy kernel and `vem` (the schematic editor) run in separate Unix processes. The Ptolemy kernel process is called “`pigiRpc`”, while the `vem` process is called “`vem`”. You can create your own version of `pigiRpc` that contains your stars and other extensions permanently linked in.

There are at least three ways to build your own `pigiRpc`, depending on the kind of extensions you are making. The first way uses `src/pigiExample`, and it is intended for users who just need to add new stars. The second and third ways use the `mkPtolemyTree` script and csh aliases and are for users that are creating new domains or making other more extensive changes.

If you want to extend Ptolemy by modifying or adding a new scheduler, target, or even an entire domain, it is recommended that you create a duplicate directory hierarchy. This allows you to experiment with and fully test any changes separately, rather than incorporating them into the “official” version of Ptolemy. This way, your experimentation will not interfere with other Ptolemy users at your site, and your changes will not be overwritten by future installations of Ptolemy releases. It also means that all of the existing makefiles will work without modification because all of the paths specified are relative to the root of the hierarchy.

The most direct way to do this is to copy the entire Ptolemy hierarchy. This could be done with a command such as:

```
cp -r $PTOLEMY ~/ptolemy
```

which would create a copy of the hierarchy in your home directory. Because this method requires excessive disk space and makes cooperative development difficult, many developers prefer to use symbolic links when creating a duplicate hierarchy. `mkPtolemyTree` and the csh aliases can help you setup these symbolic links.

1.3.1 Creating a pigiRpc that includes your own stars

For those who just want to statically link in their own stars with minimal hacking with makefiles, an example showing how to do this is provided in `$PTOLEMY/src/pigiExample.`

In the example below, we assume that `$PTOLEMY` and `$PTARCH` are set and that you have write permission to the Ptolemy source tree. If you don't have write permission, you can set up a parallel tree with the Unix `ln -s` command. If, for example, the Ptolemy tree was at `/users/ptolemy`, but you wanted to build under `~/pt`, you could do the following to create the directory and create symbolic links for the dot files, like `.cshrc`, and create symbolic links for the other files and directories in the distribution:

```
mkdir ~/pt
cd ~/pt
ln -s /users/ptolemy/* .
ln -s /users/ptolemy/.??* .
setenv PTOLEMY ~/pt
setenv PTARCH ‘$PTOLEMY/bin/ptarch‘
rm obj.$PTARCH src bin.$PTARCH
mkdir -p src src/pigiExample bin.$PTARCH
cd bin.$PTARCH; ln -s /users/ptolemy/bin.$PTARCH
cd ../src; ln -s /users/ptolemy/src/* .
cd pigiExample; cp /users/ptolemy/src/pigiExample/* .
```

You also need to be sure that you have your environment set up properly for the compiler that you are using.

Continuing with our example of how to build a `pigiRpc` that includes your own stars:

1. Build a basic `pigiRpc`. `PigiRpc` depends on `.o` files under `$PTOLEMY/obj.$PTARCH`, so you must do a basic build. To build all the `.o` files, type:

   ```
   cd $PTOLEMY; make install
   ```

   The complete build process can take upwards of three hours. If you use an `override.mk` file, you can reduce the build time by building only the functionality you need. See “Using mkPtolemyTree to create a custom Ptolemy trees” on page 1-9 for more information.

2. Edit `$PTOLEMY/src/pigiExample/make.template`. Add your stars to `LOCAL_OBJS` and `PL_SRCS`.

3. `cd` to `$PTOLEMY/obj.$PTARCH/pigiExample` and type:

   ```
   make depend
   ```

   to update the `makefile` from the `make.template`. You will see messages something like:

```
makefile remade -- you must rerun make.
exit 1
make: *** [makefile] Error 1
```

This is normal and you may safely ignore the error message.

4. While still in `$PTOLEMY/obj.$PTARCH/pigiExample`, type

   ```
   make
   ```

   This will create a version of the `pigiRpc` executable with your own stars statically linked in. If later you add a new star, you should modify the symbols `OBJS` and `PLSRCS` in `make.template` to include it, and repeat the above procedure.

5. If you built your `pigiRpc` with `SDFMyStar.o`, you can test your `pigiRpc` by starting up with:

   ```
   pigi -rpc $PTOLEMY/obj.$PTARCH/pigiExample/pigiRpc $PTOLEMY/
   src/pigiExample/init.pal
   ```

   and then run the 'wave' universe. If you want to have the binary you just built be the default binary for yourself, you can set your `PIGIRPC` environment variable to the name of the binary you just built:

   ```
   setenv PIGIRPC $PTOLEMY/obj.$PTARCH/pigiExample/pigiRpc
   ```

   Next time you start `pigi`, your new executable will be used instead of the standard one. To revert to using the installed `pigiRpc`, just type

   ```
   unsetenv PIGIRPC
   ```

6. If you want your `pigiRpc` to be the default `pigiRpc`, you can install it in `$PTOLEMY/bin.$PTARCH`, but this will wipe out whatever `pigiRpc` is in that directory

With the same makefile, you can make a version of the `pigiRpc` program that has debug symbols. Just type:

```
make pigiRpc.debug
```

To use this, assuming the Gnu debugger `gdb` is in your path, specify the executable as follows:

```
setenv PIGIRPC \
       $PTOLEMY/obj.$PTARCH/pigiExample/pigiRpc.debug
```

assuming your executable is in `$PTOLEMY/obj.$PTARCH/pigiExample/`. Then start `pigi` as follows:

```
pigi -debug
```

To revert to using the installed `pigiRpc`, just type

```
unsetenv PIGIRPC
```

1.3.2 Creating a pigiRpc with more extensive customizations

If you are extending Ptolemy in nontrivial ways, such as writing a new domain, we suggest that you create your own copy of the Ptolemy directory tree. You may use symbolic links to the “official” directories if you do not need to modify or work on them. Your new code should be placed in the parallel directory with the other similar Ptolemy subdirectories, using the same directory structure. This way you can reuse the makefiles of similar Ptolemy directories with minimal modifications. After you create your own Ptolemy tree and add your new directories and files, certain Ptolemy makefiles, typically `$PTOLEMY/mk/ptbin.mk` and `$PTOLEMY/mk/stars.mk`, need to be modified to include your own code. Building your own `pigiRpc`, `ptcl` or `tysh` this way requires extensive knowledge of the Ptolemy directory tree structure and makefiles, but if you are doing serious development in Ptolemy, you will need to know this anyway.

Warning: If you have write permission in the directory where Ptolemy is installed, make sure to modify the place where “make install” puts the completed executable, or it will attempt to overwrite the `pigiRpc` in the Ptolemy installation, and other users may be upset with you if you succeed in doing that. (If you are using the makefile from `$PTOLEMY/src/pigiExample`, you do not need to worry about this because “make install” has been removed from that makefile.) The simplest thing to do is to replace the line in the `makefile`:

```
install: makefile $(DESTBIN)
```

with:

```
install: makefile pigiRpc
```

This will leave the `pigiRpc` in whatever directory you make it even if you type:

```
make install
```

1.4 Using mkPtolemyTree to create a custom Ptolemy trees

In Ptolemy 0.6 and later, there are two methods of building custom Ptolemy trees that have a user selected set of domains: csh aliases and the `mkPtolemyTree` script. This section discusses the `mkPtolemyTree` script, see “Using csh aliases to create a Parallel Software Development Tree” on page 1-12 for an alternative method of creating a parallel tree.

In Ptolemy 0.6 and later, the `mkPtolemyTree` script and a user supplied `override.mk` file to create an entire custom object tree. The tree will have copies of all Ptolemy directories on which the customized installation depends. The script will also set up the `override.mk` files needed to build custom `pigiRpc`, `tysh` and `ptcl` binaries. Since `mkPtolemyTree` runs very fast, you may avoid having to recompile the entire Ptolemy tree, which can take 3 hours on a fast workstation.

1.4.1 mkPtolemyTree example

The `mkPtolemytree` command usage is:

`mkPtolemyTree` *`override.mk_file`* `r`*`oot_pathname_of_new_tree`*

For example, say that you wanted to build a tree that only has the VHDL domain in `~/mypt`.

1. One would create a file called `~/override.mk` that contains:

```
VHDL=1
DEFAULT_DOMAIN=VHDL
```

```
VERSION_DESC="VHDL only"
```

The file `$PTOLEMY/mk/ptbin.mk` contains a list of the makefile variables that can be set to bring in the various domains.

2. Set `$PTOLEMY` to point to the Ptolemy distribution, in this example, the Ptolemy distribution is at /users/ptolemy:

   ```
   setenv PTOLEMY /users/ptolemy
   ```

3. Set `$PTARCH` to the appropriate value:

   ```
   setenv PTARCH ‘$PTOLEMY/bin/ptarch‘
   ```

4. Set the path properly:

   ```
   set path = ($PTOLEMY/bin $PTOLEMY/bin.$PTARCH $path)
   ```

5. Execute the `mkPtolemyTree` command so that the `override.mk` file is used to create a custom tree in the `~/mypt` directory.

   ```
   mkPtolemyTree ~/override.mk ~/mypt
   ```

In general, you will want to define the variables `TK` and `HOF`. Setting `TK` indicates that you want to include Tcl/Tk extensions to the domains. Setting `HOF` means that you want to include the higher-order functions domain. The higher-order functions domain is used in many demonstrations to configure stars with multiple portholes and to specify scalable systems. So, adding these make variables in the same `override.mk` file would make it look like the following:

```
HOF=1
TK=1
VHDL=1
DEFAULT_DOMAIN=VHDL
VERSION_DESC="VHDL only"
```

1.4.2 How mkPtolemyTree works

To accumulate a list of the directories necessary to build a custom tree, `$PTOLEMY/src/stars.mk` contains a makefile variable named `CUSTOM_DIRS`. In `stars.mk`, each feature, such as `VHDL` adds directories to `CUSTOM_DIRS`. Also a feature can require sub-features, and the sub-features can add directories to `CUSTOM_DIRS`. For example, `VHDL` requires `CG`, and `CG` adds more directories to `CUSTOM_DIRS`.

When you run `$PTOLEMY/bin/mkPtolemyTree`, the following events occur:

1. From the `override.mk` file that the user specifies, the script builds a tree with the directories as specified the value of the `CUSTOM_DIRS` makefile variable.

2. Next, the files in the `$PTOLEMY` tree are copied over if the directory exists using `tar` (to save modification times).

3. For each directory specified by `CUSTOM_DIRS`, we create symbolic links to all the directories that we have not expanded from the `$PTOLEMY` tree the `make.template` and `makefile` symbolic links in the obj directories are set correctly.

4. The `override.mk` file is copied into the new tree as *`NEW_ROOT`*`/mk/override.mk`, where *`NEW_ROOT`* is the root path name of the tree we are constructing.
5. `override.mk` files are constructed that reference *`NEW_ROOT`*`/mk/override.mk` specific to `tysh`, `ptcl` and `pigiRpc`.
6. `make install` is run in *`NEW_ROOT`*`/obj.$PTARCH/` which creates the hard link for the libraries in *`NEW_ROOT`*`/lib.$PTARCH` and builds the custom `tysh`, `ptcl`, and `pigiRpc`.

This new tree has all the symbolic links and directories necessary to act as a full-fledged Ptolemy tree. You should be able to set your `PTOLEMY` environment variable to this new tree and `pigi` will run your custom `pigiRpc` binary.

Currently the Tcl libraries and Tycho are not expanded but are accessible via symbolic links. To have the utility expand the `$PTOLEMY/lib/tcl` directory, add the following line to your `override.mk` file:

```
CUSTOM_DIRS += $(CROOT)/lib/tcl
```

To expand Tycho, consult the Tycho documentation and use the `tylndir` script.

There is no documentation of the variables to pull in each domain yet. In general, it is the standard abbreviation for the domain in capital letter. For example, the Synchronous Dataflow (SDF) domain is `SDF`, the Discrete-Event (`DE`) domain is DE, and so forth. Some of the domains are split up, the entire domain can be brought in by defining *`FOO`*`FULL` (e.g., `SDFFULL` or `CGCFULL`). When defined, they include all of the SDF and CGC functionality, respectively, whereas SDF and CGC include only the basic functionality. The basic version of the SDF domain does not include the image, matrix, Matlab, DSP, and Tcl/Tk stars. If you are attempting to build a pigi that includes the Process Network (PN) domain, then you should add the following to your `override.mk` file.

```
INCLUDE_PN_DOMAIN = yes
```

For a listing of the possible make variables, refer to the `$PTOLEMY/mk/ptbin.mk` and `$PTOLEMY/mk/stars.mk` files.

1.4.3 Combining mkPtolemyTree and pigiExample

It is possible to use the `override.mk` file used by `mkPtolemyTree` in the `pigiExample` directory to create a custom `pigiRpc` with user added stars. One reason for doing this would be to that on some platforms, stars that have been incrementally linked are not visible from the debugger. Creating a custom `pigiRpc` with the star as a built in star can aid debugging.

After running `mkPtolemyTree`, edit `$PTOLEMY/src/pigiExample/make.template` and add your stars as described in “Creating a pigiRpc that includes your own stars” on page 1-7.

1.4.4 Known Bugs in mkPtolemyTree

- To build a customized pigiRpc, you set makefile variables like `SDF` or `CG56` to 1 in your `override.mk`. If you happen to have an environment variable called `SDF` or `CG56`, this procedure fails because the rule in `stars.mk` just checks whether the variable is defined or not, not what value it has. So, ensure that you have no environment variables that clash with the variables used in `override.mk`.

Suggested fix: In `stars.mk`, not only check whether a variable like `SDF` is defined, but also check its value.

Hopefully, the value is different from the other definition and the code is more robust.

- If `mkPtolemyTree` gives you the following message:

  ```
  Making a customized Ptolemy development tree using the version of
  Ptolemy installed in the directory /users/ptolemy
  The new customized Ptolemy tree will go in /users/cxh/mypt
  mkdir: illegal option -- n
  mkdir: usage: mkdir [-m mode] [-p] dirname ...
  ```

  The try setting your path so that `/usr/ucb` is before `/usr/bin`. The problem here is that in Ptolemy 0.7, the `mkPtolemyTree` script uses the `-n` option with `echo`, which is not portable.
- `mkPtolemyTree` cannot add new directories to an already existing tree, it can only be used to create a brand new parallel tree.
- `MAKEARCH` may fail when used with a tree that was created with `mkPtolemyTree`, since `MAKEARCH` may follow symbolic links into the master tree, where the user does not have write permission.
- `mkPtolemyTree` requires that the master Ptolemy tree have a fully expanded `obj.$PTARCH` directory. Otherwise you will get an error about ‘no sources rule found’.

1.5 Using csh aliases to create a Parallel Software Development Tree

Below is a set of C shell aliases that can be used to create a parallel software development tree.

1.5.1 Aliases for Managing Symbolic Links

Below are several `csh` aliases that can be helpful when managing a duplicate hierarchy that is implemented with symbolic links:

```
alias pt 'echo $cwd | sed s:${HOME}/Ptolemy:${PTOLEMY}:'
alias ptl 'ln -s `pt`/* .'
alias sw 'mv \!^ swap$$; mv .\!^ \!^; mv swap$$ .\!^'
alias exp 'mkdir .\!^; sw \!^; cd \!^; ptl'
alias rml '\rm -f `\ls -F \!* | sed -n s/@\$//p`'
alias mkl 'rml make*; ln -s `vpath`/make* .'
```

These are documented below in detail. For convenience, these aliases can be found in the file `$PTOLEMY/.alias`.

The `pt` Alias

The `pt` alias returns the name of the “official” Ptolemy directory that corresponds to the current directory, which is presumably in your personal hierarchy. This assumes that you have the environment variable `$PTOLEMY` set to the root directory of the "official" version of Ptolemy, and that your private version is in `~/Ptolemy`. If this is not the case, then you should make suitable modifications to definition of the `pt` alias. This alias is useful when you want to make a symbolic link to or otherwise access the "official" version of a file, as in

```
% cd ~/Ptolemy/src/domains/sdf/kernel
% ln -s `pt`/SCCS .
```

This will create a symbolic link in your directory `~/Ptolemy/src/domains/sdf/kernel` to the directory `$PTOLEMY/src/domains/sdf/kernel/SCCS`. (For information on source code control, see below).

The `ptl` Alias

The `ptl` alias uses the `pt` alias to create, in the current directory, symbolic links to all the files in the corresponding "official" directory. This is useful for quickly filling in the branches of a new directory in your private hierarchy.

```
% pwd
/users/me/Ptolemy/src/domains/ddf
% mkdir stars
% cd stars
% ptl
% ls -F
DDFCase.cc@            DDFLastOfN.cc@     DDFThresh.cc@
DDFCase.h@             DDFLastOfN.h@      DDFThresh.h@
DDFCase.pl@            DDFLastOfN.pl@     DDFThresh.pl@
DDFDownCounter.cc@     DDFRepeater.cc@    SCCS@
DDFDownCounter.h@      DDFRepeater.h@     TAGS@
DDFDownCounter.pl@     DDFRepeater.pl@    ddfstars.c@
DDFEndCase.cc@         DDFSelf.cc@        ddfstars.mk@
DDFEndCase.h@          DDFSelf.h@         make.template@
DDFEndCase.pl@         DDFSelf.pl@        makefile@
%
```

This creates a directory named `stars` and fills it with symbolic links to the contents of the corresponding directory in the "official" Ptolemy tree. Using the `-F` option of the `ls` command, makes it easy to see which files in a directory are symbolic links (they are marked with a trailing "@" sign).

The `sw` Alias

When experimenting with Ptolemy, you may want to switch back and forth between using the official version of some directory and your own version. You can keep two versions of the same directory (or a file). The `sw` alias swaps a file or directory *`filename`* with another file or directory *`.filename`*. The period at the beginning of the second file name makes it invisible unless you use the `-a` option of the `ls` command. For example, suppose you wish to experiment with making a change to just one file, `DDFRepeater.pl`, in the directory above, to fix a bug (and then send the bug fix back to the Ptolemy group):

```
% pwd
/users/me/Ptolemy/src/domains/ddf/stars
% sw DDFRepeater.pl
mv: cannot access .DDFRepeater.pl
% ls -a
./                      DDFEndCase.h@     DDFThresh.cc@
../                     DDFEndCase.pl@    DDFThresh.h@
.DDFRepeater.pl@        DDFLastOfN.cc@    DDFThresh.pl@
DDFCase.cc@             DDFLastOfN.h@     SCCS@
DDFCase.h@              DDFLastOfN.pl@    TAGS@
DDFCase.pl@             DDFRepeater.cc@   ddfstars.c@
DDFDownCounter.cc@      DDFRepeater.h@    ddfstars.mk@
DDFDownCounter.h@       DDFSelf.cc@       make.template@
DDFDownCounter.pl@      DDFSelf.h@        makefile@
DDFEndCase.cc@          DDFSelf.pl@
```

Notice that `DDFRepeater.pl` was moved to `.DDFRepeater.pl`. You can now create your own version of `DDFRepeater.pl`. To later reinstate the official version (e.g., you discovered that what you thought was a bug was in fact a feature),

```
% sw DDFRepeater.pl
```

The `exp` Alias

When starting your experimentation, the job of creating the parallel tree can be rather tedious. The `exp` aliases combines the functions of the `ptl` and `sw` aliases into one, making the common task of expanding a branch in the directory hierarchy easy. Suppose you type:

```
% exp stars
```

This is equivalent to the following sequence of commands:

```
% mkdir .stars
% sw stars
% cd stars
% ptl
```

Note that the command leaves you in the new directory ready to issue another `exp` command. For example, to create a duplicate of the directory `$PTOLEMY/src/domains/ddf/stars`, creating all subdirectories as you go, and linking to all the appropriate files in the Ptolemy tree,

```
% cd ~/Ptolemy
% exp src
% exp domains
% exp ddf
% exp stars
```

The `rml` Alias

The `rml` alias removes symbolic links in the current directory. Without an argument, it removes all the visible symbolic links. Any arguments are passed on to the `ls` command. So, to remove `all` symbolic links, including those that are invisible, use the `-a` option:

```
% rml -a
```

You can also give file names as arguments to remove just some of the symbolic links:

```
% rml *.o
```

The `mkl` alias

Suppose you wish to compile your change to the `DDFRepeater.pl` file, as above. You will need to make an object tree. Assume you are on a Sun Solaris 2.x platform. You have created a parallel tree already in `~/Ptolemy/src` (i.e. `~Ptolemy/src/domains/ddf/stars` exists). Create the corresponding object tree:

```
% cd ~Ptolemy
% exp obj.sol2
% exp domains
% exp ddf
% exp stars
% pwd
/users/me/Ptolemy/obj.sol2/domains/ddf/stars
```

The directory in which you are now located contains symbolic links to the `.o` files and makefiles in the official Ptolemy tree. If you run `make` here, your replacement `DDFRepeater.pl` star will be compiled in place of the official one. If you run “`make install`”, then a library will be created and installed in the directory `~/Ptolemy/lib.sol2`, assuming this directory exists.

Running `make` as above uses the makefiles in the official Ptolemy tree, because you have symbolic links to them. Suppose you wish to modify the `make.template` file in `~/Ptolemy/src/domains/ddf/stars`. In this case, you should run the `mkl` alias to replace the `makefile` symbolic links. If you have followed the above steps, try this:

```
% pwd
/users/me/Ptolemy/obj.sol2/domains/ddf/stars
% ls -F
DDFCase.o@            DDFRepeater.o@        libddfstars.a@
DDFDownCounter.o@     DDFSelf.o@            make.template@
DDFEndCase.o@         DDFThresh.o@          makefile@
DDFLastOfN.o@         ddfstars.o@
```

(This assumes that the “official” Ptolemy has been rebuilt after being installed, otherwise the .o and .a files will be missing). Expand the makefile symbolic links:

```
% ls -l make*
```

```
lrwxrwxrwx  1 eal            56 Jul 14 11:30 make.template -> /users/
ptolemy/obj.sol2/domains/ddf/stars/make.template
lrwxrwxrwx  1 eal            51 Jul 14 11:30 makefile -> /users/
ptolemy/obj.sol2/domains/ddf/stars/makefile
```

Note that they point to the “official” makefiles. To make them point to the versions in your own tree,

```
% mkl
% ls -l make*
lrwxrwxrwx  1 eal            47 Jul 14 11:31 make.template -> ../../
../../src/domains/ddf/stars/make.template
lrwxrwxrwx  1 eal            42 Jul 14 11:31 makefile -> ../../../../
src/domains/ddf/stars/makefile
```

Now you can modify the `make.template` file in your own tree as you need.

Warning

Note that modifying Ptolemy files is risky. You will have essentially created your own version of Ptolemy. You will not be able to install future releases of Ptolemy without abandoning your version. However, if you have modifications that you believe are valuable, please communicate them to the Ptolemy group at `ptolemy@eecs.berkeley.edu`. The Ptolemy group welcomes suggestions for changes.

1.5.2 Creating a Duplicate Hierarchy

Let’s look at a complete example to see how these aliases can be used. Suppose you want to modify an existing file that is part of the kernel for the SDF domain. You will need a private copy of the file that is writable. This allows you to make your changes without affecting the “official” version of Ptolemy. In order to test your change, you will have to build a private version of the interpreter `ptcl` or the graphical interface `pigiRpc`.

First, create the root directory for your duplicate hierarchy.

```
% mkdir ~/Ptolemy
```

Then go into that directory and create symbolic links to all files in the corresponding “official” Ptolemy directory.

```
% cd ~/Ptolemy
% ptl
```

You will want to have a private version of the `lib.$PTARCH` directory so that you won’t modify the “official” version of any library or object files.

```
% cd ~/Ptolemy
% exp lib.$PTARCH
```

(This assumes your `$PTARCH` environment variable is set). You will also want a private obj.$PTARCH directory for the same reason. In this example, the tree is expanded down to the sdf directory:

```
% cd ~/Ptolemy
% exp obj.$PTARCH
% exp domains
% exp sdf
```

If you are modifying code in the sdf/kernel directory, then you will want to expand it as well. Once expanded, you will want remove the make.template and makefile links (which point to the "official" Ptolemy files) and replace them with links that use relative paths to refer to your private versions of these files (in case you make changes to them):

```
% exp kernel
% mkl
```

If you make changes in the sdf/kernel directory, then there is a good chance that object files in sdf/dsp and other directories will also have to be recompiled. Thus, you will want to expand these directories (and any subdirectories below them) as well. Remember to replace the make.template and makefile links as in the sdf/kernel directory.

```
% exp dsp
% mkl
% exp stars
% mkl
```

Because of the way symbolic links work, it is important to remove the links for the .o and .a files in the directories you have just created. You can do this by issuing a make realclean command in the obj.$PTARCH/domains/sdf directory. This will recursively clean out all the subdirectories. You could also do this manually by issuing a rml *.o *.a command in each directory.

You will also need a private version of the src directory.

```
% cd ~/Ptolemy
% exp src
% exp domains
% exp sdf
% exp kernel
```

At any point after this, it is possible to switch back and forth between private and "official" versions of these directories with the sw alias. In fact, you just used it (as part of the exp alias) to switch to the private versions of the obj.$PTARCH, lib.$PTARCH, and src directories.

To compile your version of the sdf kernel directory,

```
% cd ~/Ptolemy/obj.$PTARCH/domains/sdf/kernel
% make install
```

To make a version pigiRpc (or better yet, ptinyRpc) with your changes,

```
% cd ~/Ptolemy/obj.$PTARCH
% exp pigiRpc
% mkl
% make ptinyRpc
```

1.5.3 Source Code Control

At the present time, at Berkeley, the Ptolemy group uses SCCS for source code control. This means that each directory with source code in it contains a subdirectory called SCCS. That subdirectory is not distributed with Ptolemy, but if you are starting your own development expanding on Ptolemy, you may wish to use a similar mechanism. We assume here that you are familiar with SCCS, which is a standard Unix facility.

Recall the command above:

```
% pwd
/users/me/Ptolemy/src/domains/ddf/stars
% sw DDFRepeater.pl
mv: cannot access .DDFRepeater.pl
% ls -a
./                      DDFEndCase.h@       DDFThresh.cc@
../                     DDFEndCase.pl@      DDFThresh.h@
.DDFRepeater.pl@        DDFLastOfN.cc@      DDFThresh.pl@
DDFCase.cc@             DDFLastOfN.h@       SCCS@
DDFCase.h@              DDFLastOfN.pl@      TAGS@
DDFCase.pl@             DDFRepeater.cc@     ddfstars.c@
DDFDownCounter.cc@      DDFRepeater.h@      ddfstars.mk@
DDFDownCounter.h@       DDFSelf.cc@         make.template@
DDFDownCounter.pl@      DDFSelf.h@          makefile@
DDFEndCase.cc@          DDFSelf.pl@
```

Note the symbolic link to the “official” SCCS directory. This will not be present if you are using the distributed Ptolemy and have not created it. Assume, however, that you have put this directory under SCCS control (or someone else has). Then you can create an editable version of the `DDFRepeater.pl` star with the command:

```
% sccs edit DDFRepeater.pl
1.24
new delta 1.25
76 lines
```

The sccs utility tells you the latest version number (1.24) and assigns you a new version number (1.25). You can now edit the file safely (nobody else will be allowed by sccs to edit it). When you are done and have fully tested your changes (and obtained clearance from the Ptolemy group if necessary), you can check the file back in:

```
% sccs delget DDFRepeater.pl
comments?
```

You should enter an explanation of your changes. If you wish to nullify your changes, restoring the official version,

```
% sccs unedit DDFRepeater.pl
```

and if you wish to create a new file and put it under SCCS control,

```
% sccs create -fi NewFileName
```

1.6 Building standalone programs that use Ptolemy libraries.

Sometimes it is necessary to create small standalone programs that use part of the Ptolemy libraries.

Examples of this are the desire to use Ptolemy kernel classes such as `StringList` or the need to isolate an obscure bug or memory leak. The `$PTOLEMY/mk/standalone.mk` file provides the make definitions to make this possible. This file provides make rule definitions to build various binaries some using the Pure Sofware Inc.[1] utilities.

The usage for this makefile is:

`make -f $PTOLEMY/mk/standalone.mk` *`stars.mk_variable_defs filename`*`.`*`suffix`*

Where *`stars.mk_variable_defs`* is zero or more makefile variables used in `$PTOLEMY/mk/stars.mk`, such as `SDF=1`. *`filename`* is the base name of the file to be compiled, and the basename of the output file. and *`suffix`* is one of the forms listed in table 1-1.

Suffix	Binary Type
`.bin`	Standard binary
`.debug`	Binary with debug symbols
`.purify`	Binary with Purify and debug symbols
`.quantify`	Binary with Quantify linked in
`.purecov`	Binary with Pure Coverage linked in

TABLE 1-1: Table of filename suffixes and binary types.

It is possible to use these makefiles to create binaries that do not have any Ptolemy code. A reason why you might want to do this is to take advantage of the Pure Software make definitions in `standalone.mk`. To specify no Ptolemy libraries, use the make argument `NOPTOLEMY=1`.

1.6.1 Standalone example using StringList

For example, say you want to use the `StringList` class in a standalone program

1. Rational (http://www.rational.com) sells tools such as:
Purify, which can be used to find memory leaks and out of bounds memory accesses.
Quantify, which can be used to profile performance.
Purecov, which can be used to provide code coverage information.

named `bar.cc`:

```
      #include
      #include "StringList.h"
      main() {
            StringList testing = "This is a test\n";
            cout << testing;
      }
```

To build it you would type:

```
   make -f $PTOLEMY/mk/standalone.mk bar.bin
```

If you wanted to make a new standalone program that also uses part of the CG domain, just define the domain make variables (as used in stars.mk) on the make command line:

```
   make -f $PTOLEMY/mk/standalone.mk CG=1 bar.bin
```

If you are going to do this often, it may be useful to create a new directory in which to test this program. In this directory, execute the commands:

```
   ln -s $PTOLEMY/mk/standalone.mk makefile
   ln -s $PTOLEMY/mk/standalone.mk make.template
```

By having these symbolic links, you will not have to supply the make argument `-f $PTOLEMY/mk/standalone.mk` as before.

1.6.2 Standalone example that tests a Scheduler

Here is an example of a minimal file that can be used to call the setup in a Scheduler for instance. If the file `testAcyLoopSched.cc` contains:

```
      #include <iostream.h>
      #include "Galaxy.h"
      #include "SDFStar.h"
      #include "AcyCluster.h"
      #include "AcyLoopScheduler.h"
      #include "SDFPortHole.h"

      main() {
            // First create a simple galaxy and some stars.
            SDFStar star[3];
            Galaxy topGalaxy;
            topGalaxy.setDomain("SDF");
            topGalaxy.setName("topGalaxy");
            topGalaxy.addBlock(star[0],"star0");
            topGalaxy.addBlock(star[1],"star1");
            topGalaxy.addBlock(star[2],"star2");

            // Add ports to stars.
            OutSDFPort p0,p1;
            InSDFPort p2,p3;

            // initialize the ports
            p0.setPort("output1",&star[0],FLOAT,2);
            star[0].addPort(p0);
            p1.setPort("output2",&star[0],FLOAT,3);
            star[0].addPort(p1);
            p2.setPort("input",&star[1],FLOAT,3);
```

```
        p3.setPort("input",&star[2],FLOAT,2);
        star[1].addPort(p2);
        star[2].addPort(p3);

        // Connect ‘em up.  The graph is
        // star[1] (3) <--- (2) star[0] (3) ---> (2) star[2]
        p0.connect(p2,0);
        p1.connect(p3,0);

        // Scheduling
        AcyLoopScheduler sched;
        sched.setGalaxy(topGalaxy);
        cout << "No problem till now.  Calling sched.setup()...\n";
        sched.setup();
        int i;
        for (i = 0 ; i < 3 ; i++) {
              cout << star[i].fullName() << "\n";
              cout << "Repetitions = " << star[i].reps() << "\n";
        }
        StringList sch = sched.displaySchedule();
        cout << sch;
}
```

The command to compile this and produce a standalone binary would be:

```
make -f $PTOLEMY/mk/standalone.mk OPTIMIZER= SDF=1 \
      USE_SHARED_LIBS=yes testAcyLoopSched.debug
```

1.7 Debugging Ptolemy and Extensions Within Pigi

The extensibility of Ptolemy can introduce problems. Code that you add may be defective (few people write perfect code every time), or may interact with Ptolemy in unexpected ways. These problems most frequently manifest themselves as a Ptolemy crash, where the Ptolemy kernel aborts, creating a core file.

The fact that `pigiRpc` and `vem` are separate Unix processes has the advantage that when `pigiRpc` aborts with a fatal error, `vem` keeps running. Your `vem` schematic is unharmed and can be safely saved. Vem gives a cryptic error message something like:

```
RPC Error: server: application exited without calling
RPCExit
Closing Application /home/ohm1/users/messer/ptolemy/lib/
pigiRpcShell on host foucault.berkeley.edu
Elapsed time is 1538 seconds
```

The message

```
segmentation fault (core dumped)
```

may appear in the window from which you started `pigi`. The first line in the above message might alternatively read

```
RPC Error: fread of long failed
```

Vem is trying to tell you that it is unable to get data from the link to the Ptolemy kernel. In either case, it will create a large file in your home directory called `core`. The `core`[1] file is useful for finding the problem.

1.7.1 A quick scan of the stack

Assuming you are using Gnu tools, and assuming the `pigiRpc` executable that you are using is in your path, go to your home directory and type:

```
gdb pigiRpc
```

The Gnu symbolic debugger (`gdb`) will show the state of the stack at the point where the program failed. Note that `gdb` is not distributed with Ptolemy, but is available free over the Internet in many places, including `ftp://prep.ai.mit.edu/pub/gnu`. The most recently called function might give you a clue about the cause of the problem. Here is a typical session:

```
cxh@watson 197% gdb pigiRpc ~/core
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type “show copying” to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for
details.
GDB 4.15.1 (sparc-sun-solaris2.4),
Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
```

Tell `gdb` to read in the core file.

```
(gdb) core core
Core was generated by `/users/ptolemy/bin.sol2/pigiRpc :0.0 wat-
son.eecs.berkeley.edu 32870 inet 1 2 3’.
Program terminated with signal 11, Segmentation fault.
Reading symbols from
            /users/ptolemy/lib.sol2/libcg56dspstars.so...done.
Reading symbols from
            /users/ptolemy/lib.sol2/libcg56stars.so...done.
```

Since this version of Ptolemy uses shared libraries, we see lots of messages about shared libraries, which we’ve deleted here for brevity.

```
(gdb) where
#0  0xee7a1c20 in _kill ()
#1  0x52b04 in pthread_clear_sighandler ()
#2  0x52cb4 in pthread_clear_sighandler ()
#3  0x53130 in pthread_clear_sighandler ()
#4  0x53320 in pthread_handle_one_process_signal ()
#5  0x55658 in pthread_signal_sched ()
#6  0x554d8 in called_from_sighandler ()
#7  0x535e4 in pthread_handle_pending_signals ()
#8  0x10100c in SimControl::getPollFlag ()
#9  0x101604 in Star::run ()
#10 0xd394c in DataFlowStar::run ()
#11 0xeeca5fb8 in SDFAtomCluster::run (this=0x2bd0b0)
at ../../../../src/domains/sdf/kernel/SDFCluster.cc:1032
#12 0xeeca0f20 in SDFScheduler::runOnce (this=0x2bd050)
at ../../../../src/domains/sdf/kernel/SDFScheduler.cc:121
#13 0xeeca0eac in SDFScheduler::run (this=0x2bd050)
at ../../../../src/domains/sdf/kernel/SDFScheduler.cc:98
```

1. Note that core files can be large in size, so your system administrator may have setup the csh “limit” command to disable the creation of core files. For further information, see the csh man page.

```
#14 0x108358 in Target::run ()
#15 0x109e04 in Runnable::run ()
#16 0xe62ec in InterpUniverse::run ()
#17 0xee9e7f04 in PTcl::run (this=0x20af80, argc=2949528,
argv=0x109fa4)
at ../../src/ptcl/PTcl.cc:521
#18 0xee9e99a4 in PTcl::dispatcher (which=0x27, interp=0x1d4830,
argc=2,
```

The "where" command shows that state of the stack at the time of the crash. The actual stack trace was 72 frames long, the last two frames being:

```
#71 0xeec06d5c in ptkMainLoop ()
     at ../../src/pigilib/ptkTkSetup.c:192
#72 0x4982c in main ()
```

Scanning this list we can recognize that the crash occurred during the execution of a star. Unfortunately, unless you are running a version of `pigiRpc` with the debug symbols loaded, it will be difficult to tell much more from this.

1.7.2 More extensive debugging

To do more extensive debugging, you need to create or find a version of `pigiRpc` with debug symbols, called `pigiRpc.debug`.

The first step is to build a `pigiRpc` that contains the domains you are interested in debugging. There are several ways to build a `pigiRpc`:

a. There may be prebuilt debug binaries on the Ptolemy Web site, check the directory that contains the latest release.

b. Rebuild the entire tree from scratch. This takes about 3 hours. Appendix A in the Ptolemy User's Manual has instructions about this.

c. Use `mkPtolemyTree` to rebuild a subset of the Ptolemy tree. See "Using mkPtolemyTree to create a custom Ptolemy trees" on page 1-9 for more information.

d. Use the csh aliases to rebuild a subset of the Ptolemy tree. See "Using csh aliases to create a Parallel Software Development Tree" on page 1-12 for more information.

The next step is to build the `pigiRpc.debug` binary:

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

Then set the `PIGIRPC` environment variable to point to the binary:

```
setenv PIGIRPC $PTOLEMY/obj.$PTARCH/pigiRpc/pigiRpc.debug
```
[1]

Then run pigi as follows:

```
pigi -debug
```

An extra window running `gdb` appears. (If this fails, then `gdb` is probably not installed at your

1. Note that the pigi script will attempt to find pigiRpc.debug binary if the PIGIRPC environment variable is not set. An alternative is that one can avoid setting PIGIRPC and use the pigi -rpc option to specify a binary.The command would be:
pigi -debug -rpc $PTOLEMY/obj.$PTARCH/pigiRpc/pigiRpc.debug

site or is not in your path.) Type `cont` to continue past the initial breakpoint.

Now, if you can replicate the situation that created the crash, you will be able to get more information about what happened. Here is a sample of interaction with the debugger through the gdb window:

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type “show copying” to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for
details.
GDB 4.15.1 (sparc-sun-solaris2.4),
Copyright 1995 Free Software Foundation, Inc...
Breakpoint 1 at 0x39ab4: file ../../src/pigiExample/pigiMain.cc, line
58.
Breakpoint 1, main (argc=-282850408, argv=0x399c0)
at ../../src/pigiExample/pigiMain.cc:58
58              pigiFilename = argv[0];
(gdb) cont
Continuing.
```

At this point, you are running Ptolemy. Use it in the usual way to replicate your problem. When you succeed, you will get a message something like:

```
Program received signal SIGSEGV, Segmentation fault.
0xeee81394 in mxRealMax ()
(gdb)
```

At this point you can again examine the stack. This time, however, there will be more information. Here, we examine the top 5 frames of the stack

```
(gdb) where 5
#0  0xeee81394 in mxRealMax ()
#1  0xe3864 in SimControl::getPollFlag () at ../../src/kernel/SimCon-
trol.cc:271
#2  0xe3e5c in Star::run (this=0x28c908) at ../../src/kernel/
Star.cc:73
#3  0xbacb8 in DataFlowStar::run (this=0x28c908)
at ../../src/kernel/DataFlowStar.cc:94
#4  0xef485fb8 in SDFAtomCluster::run (this=0x278570)
at ../../../../src/domains/sdf/kernel/SDFCluster.cc:1032
(More stack frames follow...)
(gdb)
```

This particular stack trace is a little strange at the “bottom” (gdb calls the lower numbers the bottom even though they are at the top of the list) because it was generated by invoking a dynamically linked star, and the symbol information is not complete. However, you can still find out quite a bit. Notice that you are now told where the files are that define the methods being called. The file names are all relative to the directory in which the corresponding object file normally resides. The Ptolemy files can all be found in some subdirectory of `$PTOLEMY/src`.

You can get help from gdb by typing “help”. Suppose you wish to find out first which star is being run when the crash occurs. The following sequence moves up in the stack until the “run” call of a star:

```
(gdb) up
#1  0xe3864 in SimControl::getPollFlag () at ../../src/kernel/SimCon-
trol.cc:271
```

```
271             ptBlockSig(SIGALRM);
(gdb) up
#2  0xe3e5c in Star::run (this=0x28c908) at ../../src/kernel/
Star.cc:73
73              go();
(gdb)
```

At this point, you can see that line 73 of the file `$PTOLEMY/src/kernel/Star.cc` reads

```
go();
```

Odds are pretty good that the problem is in the `go()` method of the star. You can find out to which star this method belongs as follows:

```
(gdb) p *this
$1 = {<Block> = {<NamedObj> = {nm = 0x28ad58 "BadStar1",
      prnt = 0x28c878,
      myDescriptor = 0x28b658 "Causes a core dump deliberately",
      _vptr. = 0xeee91738}, flags = {nElements = 0, val = 0x0},
      pTarget = 0x28aa60, scp = 0x0,
ports = {<NamedObjList> = {<SequentialList> =
      {lastNode = 0x0, dimen = 0}, }, }, states = {<NamedObjList> =
      {<SequentialList> = { lastNode = 0x0, dimen = 0}, }, },
multiports = {<NamedObjList> = {<SequentialList> =
      {lastNode = 0x0, dimen = 0}, }, }},
      indexValue = -1, inStateFlag = 1}
(gdb)
```

This tells you that a star with name (`nm`) `BadStar1` and descriptor “Causes a core dump deliberately.” is being invoked. This particular star has the following erroneous go method:

```
go {
      char* p = 0;
      *p = ’c’;
}
```

More elaborate debugging requires that the symbols for the star be included. The easiest way to do this is to build a version of `pigiRpc.debug` that includes your star already linked into the system. Then repeat the above procedure. The bottom of the stack frame will have much more complete information about what is occurring.

1.7.3 Debugging hints

Below are some hints for debugging.

- “Using emacs, gdb and pigi” on page 1-26
- “Gdb and the environment” on page 1-26
- “Optimization” on page 1-26
- “Debugging StringLists in gdb” on page 1-26
- “How to use ptcl to speed up the compile/test cycle.” on page 1-27

- “Miscellaneous debugging hints for gdb” on page 1-28

  See also Appendix A of the Ptolemy User’s manual.

Using emacs, gdb and pigi

By default, `gdb` is started in an X terminal window with its default command line interface. Many people prefer to interface with `gdb` through `emacs`, which provides much more sophisticated interaction between the source code and the debugger. To get an `emacs` interface to `gdb` (assuming `emacs` is installed on your system), set the following environment variable:

```
setenv PT_DEBUG ptgdb
```

To find out more about using `gdb` from within `emacs`, start up `emacs` and type:

```
M-x info
Then type:
m emacs
```

Then go down to:

```
Running Debuggers Under Emacs

* Starting GUD::        How to start a debugger subprocess.
* Debugger Operation::  Connection between the \
debugger and source buffers.
* Commands of GUD::     Key bindings for common commands.
* GUD Customization::   Defining your own commands for GUD.
```

Gdb and the environment

Note that the documentation for `gdb` says the following:

```
*Warning:* GDB runs your program using the shell indicated by your
`SHELL' environment variable if it exists (or `/bin/sh' if not). If
your `SHELL' variable names a shell that runs an initialization file-
-such as `.cshrc' for C-shell, or `.bashrc' for BASH--any variables
you set in that file affect your program. You may wish to move setting
of environment variables to files that are only run when you sign on,
such as `.login' or `.profile'.
```

Optimization

By default, Ptolemy is compiled with the optimizer turn up to a very high level. This can result in strange behavior inside the debugger, as the compiler may evaluate instructions in a different order than they appear in the source file. You may find it easier to debug a file by recompiling it with the optimization turned off by removing the corresponding `.o` file and doing:

```
make OPTIMIZER= install
```

Debugging StringLists in gdb

Ptolemy uses `StringList` object to manipulate strings. However, using `gdb` to view a `StringList` object can be non-intuitive. To print the contents of a `Stringlist` *`myStringList`* as one item per line from within `gdb`, use:

```
p displayStringListItems(myStringList)
```

To print out the `StringList` as a contiguous string, use:

```
p displayStringList(myStringList)
```

How to use ptcl to speed up the compile/test cycle.

If you are spending a lot of time debugging a problem, you may want to use `ptcl` instead of `pigiRpc`, as `ptcl` is smaller and starts up faster. Also, you can keep your breakpoints between invocations of `ptcl`, as debugging `ptcl` does not start up a separate `emacs` each time. However, `ptcl` cannot handle demos that use Tk.

Here's how to use `ptcl` to debug.

1. Run `pigiRpc` on the universe, and use compile-facet to generate a `~/pigiLog.pt` file. Note the number of iterations for the universe, and then exit `pigiRpc`.
2. Copy `~/pigiLog.pt` to somewhere. A short file name, like `/tmp/tst.tcl` will save time in typing since you may be typing it often. Don't use something inside your home directory as you can't easily use ~ inside `ptcl`.
3. Edit the file and add a `run` *`XXX`* line and a `wrapup` line at the end. If the demo should run for 100 iterations, then add:

   ```
   run 100
   wrapup
   ```

   to the end of the file.
4. Build a `ptcl.debug` that has just exactly the functionality you need by using an `override.mk` file. Alternatively, you could use either `ptcl.ptrim.debug` or `ptcl.ptiny.debug`. If your demo is SDF, then try building and using `ptcl.ptiny.debug`.
5. If you use `emacs`, then you can start up `gdb` on your binary with:

   ```
   M-x gdb
   ```
6. Then type in the name of the binary. You may have to use the full pathname.
7. Inside `emacs`, you can then set breakpoints in the `gdb` window, either by typing a break command, or by viewing the file and typing `Control-X space` at the location you would like a break point.
8. Type `r` to start the process, and then source your demo with:

   ```
   source /tmp/tst.tcl
   ```

   If you want to recompile your demo outside of `gdb` and then reload it into your `gdb` session, use the `file` command inside `gdb`:

```
file /users/cxh/pt/obj.sol2/ptcl/ptcl.ptiny.debug
```

Your breakpoints will be saved, which is a big time saver.

Miscellaneous debugging hints for gdb

If you are having problems debugging with `gdb`, here’s what to check.

1. Verify that your `$PTOLEMY` is set to what you intended. If you are building binaries in your private tree, be sure that `$PTOLEMY` is set to your private tree and not `~ptdesign` or `/users/ptolemy`.
2. Verify that your `$LD_LIBRARY_PATH` does not include libraries in another Ptolemy tree. You could type:

   ```
   unsetenv $LD_LIBRARY_PATH
   ```

3. `gdb` sources your `.cshrc`, so your `$PTOLEMY` and `$LD_LIBRARY_PATH` could be different. Inside `gdb`, use

   ```
   show env PTOLEMY
   ```

   to see what it is set to. This problem is especially common if you are running `gdb` inside `emacs` via `ptgdb`.
4. Verify that you are running the right binary by looking at the creation times. You may find it useful to use the `-rpc` option:

```
pigi -debug -rpc $PTOLEMY/obj.$PTARCH/pigiRpc/pigiRpc.mine ~ptdesign/
init.pal
```

5. Recompile the problem files with optimization turned off and relink your `pigiRpc`. You can do this with

```
rm myfile.o; make OPTIMIZER= install
```

   Then rebuild your `pigiRpc`

6. Look for weird coding styles that could confuse the line count in emacs and gdb, such as declaring variables in the middle of a block and brackets that open a function body on the same line as the function declaration:

```
int foo(int bar){
```

   vs.

```
int foo(int bar)
            {
```

7. Use `stepi` to step by instructions, rather than `step`.