1. An Overview of Ptolemy

Authors:

Joseph Buck
Edwin E. Goei
Soonhoi Ha
Alan Kamas
Ichiro Kuroda
Phil Lapsley
Edward A. Lee
David G. Messerschmitt

1.1. Introduction

Ptolemy is a very flexible foundation upon which to build simulation environments, where a key objective is the ability to combine these environments into multi-paradigm simulations as necessary. To effectively use existing simulation and synthesis tools and methodologies, different models of computation must be used for different parts of the overall design. Large systems often mix hardware, software, and communication subsystems. They may also combine hardware targets, including custom logic, processors with varying degrees of programmability, systolic arrays, and other multiprocessor subsystems. Tools supporting each of these components are different, using for instance dataflow principles, regular iterative algorithms, communicating sequential processes, control/dataflow hybrids, functional languages, and discrete-event system theory and simulation.

Ptolemy is a third-generation software environment that supports heterogeneous system specification, simulation, and design. It is an object-oriented framework within which diverse models of computation can co-exist and interact. In addition to the usual use of hierarchy to manage complexity, **Ptolemy** uses hierarchy to mix heterogeneous models of computation. The result is a unified software environment that extends the philosophy of mixed-mode circuit simulation up to the design and simulation of complex systems.

This work is an outgrowth of two previous generations of design environments, Blosim [Mes84a,b] and Gabriel [Bie90][Lee89b], that were aimed at digital signal processing applications. Both environments used a block-diagram data-flow paradigm for the description of the algorithms. To broaden the applicability beyond DSP, we see the need for other computational models, such as discrete-event scheduling, mixed compile-time and run-time scheduling, or computational models based on shared data structures. These are not supported very gracefully by Blosim or Gabriel. Most importantly, we see the need for a flexible simulation environment which is extensible to new computational models without re-implementation of the system.

Ptolemy uses an object-oriented programming methodology to support heterogeneity, and is programmed in C++. Data abstraction and polymorphism, two tenets of object-oriented programming, allow models of computation to be abstracted so that their differences are not visible from other domains. Our goal is to make the system non-dogmatic, in the sense that the environment itself does not impose any particular computational model, and it is extensible to new models by simply adding to the system and not modifying what is already there. The overall software architecture is described in [Buc92b]. Further goals are to incorporate features that have been successful in Blosim or Gabriel, such as achieving modularity and reusability of user-programmed software modules, friendly graphical window interfaces, and code generation for targeting concurrent architectures rather than just simulation.

Each model of computation in **Ptolemy** is called a `Domain.` The system currently has a synchronous data-flow (SDF) domain [Bha92] [Buc91] [Lee87a] [Lee87b] [Lee91b] (like Gabriel), a dynamic dataflow (DDF) domain [Ha91] [Ha92] (like Blosim), a discrete-event (DE) domain, a Thor domain [Tho86] (for circuit simulation) and various code generation domains using dataflow semantics [Pin92]. We are working on several more domains that have not reached sufficient maturity to be included in this release [Buc92a] [Lee91a] [Lee92a] [Lee93]. The system is, however, already capable of simulating combinations of signal processing and communication networks (such as in packet speech and packet video) and combinations of behavioral and hardware simulation [Kal91] [Kal92], and is capable of synthesizing real-time implementations in assembly code for Motorola DSPs, as well as accelerated simulation code in C [Pin92].

The graphical interface (pigi) is based on VEM [Har86], a graphical editor for the OCT design database [Har86], both developed in the CAD group at Berkeley.

A word about notation: In this and other **Ptolemy** documentation, keywords from C++, or an abstract data type defined as a `class` in C++, or fragments of C++ code, are printed in the special font just used for "class". However, since the documentation is written by many people, we cannot promise perfect consistency.

1.2. Terminology

Ptolemy relies on a basic very flexible computational model of a simulation. The overall simulation is decomposed into software modules called *blocks*. These blocks at runtime are invoked in an order determined by a *scheduler*, and exchange data among themselves as they execute. From the user perspective there are two types of blocks: the `Star` and the `Galaxy`. The `Star` is elemental, in the sense that it is implemented by a user-provided code. There are also many pre-coded `Star`s in the **Ptolemy** library, but these should be viewed as examples, rather than as a comprehensive set. Adding new blocks is easy, so the system should be viewed as a programming environment, and not just as a monolithic tool to be used unmodified. A `Galaxy` is a block which internally contains `Star`s as well as possibly other `Galaxy`s. The `Galaxy` is thus a construct for producing a hierarchical description of the simulation. A `Universe` is a complete program, or application.

One of the key innovations in **Ptolemy** is the extension of the hierarchy of stars, galaxies, and universes to include objects called `Wormhole`s. A wormhole is named as such because from the outside, it looks monolithic, like a star, but inside, it looks more like a universe. The scheduler on the outside treats it exactly like a star, but internally it contains its own scheduler. The inside and outside schedulers need not abide by the same model of computation. This is the basic mechanism for supporting heterogeneity.

Data passes between blocks in discrete units called `Particle`s. For example, a particle in a signal processing system is often a signal sample, usually a floating-point value. But it can also be an image, for example, in a video sequence. In a communication system, it may be a packet consisting of multiple fields. In domains using dataflow principles, a particle may be called a *token*. Particles pass from one domain to another (into or out of a wormhole) through an

`EventHorizon.` The event horizon manages any format translations that may be required to stitch together two models of computation. User-defined particles are supported.

To get started, you can use the **Ptolemy** interactive graphical interface (pigi), described below, or the **Ptolemy** interpreter (ptcl). The interpreter is based on a language called Tcl, invented at Berkeley [Ous90]. Thus, both a graphical and textual specification language are available. It is unlikely that these two user interfaces will be the only ones developed for **Ptolemy**. They are, in fact, quite distinct from the **Ptolemy** kernel, precisely so that other types of user interfaces can be readily accepted. Future versions of the graphical interface will also use Tcl and its associated X11 toolkit, Tk [Ous91], which will lend it much more flexibility.

1.3. Installation

Ptolemy is a large software system that relies on a properly configured software environment. There are many things that can go wrong in getting **Ptolemy** running. For instance, the windowing system may not be the same one we use, or it may be used in a different way. There is also some configuration required by each user in order to use the graphical interface. The information for doing this is given in the Pigi document, section 2 of the Almagest. Here we give the basic information required to get from an FTP archive or distribution tape to being able to run the system.

1.3.1. Basic Ptolemy installation

First note that the approximate disk space requirements are (for the Sun-4 distribution; other distributions are roughly the same size):

Ptolemy: 49 Mbytes
Ptolemy (after you optionally remake): 64 Mbytes
Gnu subset: 16 Mbytes

Create a user ‘‘ptolemy’’, together with a home directory for the ‘‘ptolemy’’ user. Once the "ptolemy" user account has been created, log in or su to user ‘‘ptolemy’’. If you do not wish to create a user called ‘‘ptolemy’’, see below for an alternative.

If you are loading Ptolemy from a tape, do the following:

a. cd /tmp (or any directory where you have write permission and there is at least 22 Mbytes of free disk space).

b. Load the tape into your drive

c. mt -f /dev/nrst8 rewind
(This rewinds the tape -- change the device name if your tape drive has a different name.)

d. tar xf /dev/nrst8

If you have used FTP to down load the files, then cd to the directory that contains the "*tar.Z" files you downloaded via FTP.

Now, whether you’ve used tape or FTP, there should be a number of large "*tar.Z" files in your current directory. Proceed as follows:

1. zcat pt-0.4.doc.tar.Z | (chdir ˜ptolemy/..; tar xf -)
(this uncompresses the documentation, changes directory the parent of the "ptolemy" user, and then creates all of the documentation files.)

2.
   ```
   zcat pt-0.4.src.tar.Z |  ( chdir ~ptolemy/..; tar xf - )
   ```
   (this uncompresses the src and creates the source files.) You must not skip this step. **Ptolemy** depends on these files being present. Note that you may get a few warning messages during this and the following step about the tar program not being able to create some directories because they already exist. This is expected (the same directory is mentioned in several of the tar files), so you need not worry.
3. If you are running the DecStation (MIPS) version of **Ptolemy**:
   ```
   zcat pt-0.4.mips.tar.Z |  ( chdir ~ptolemy/..; tar xf - )
   ```
   If you are running the Sun 4 (Sparc, IPC, IPX, etc) version:
   ```
   zcat pt-0.4.sun4.tar.Z |  ( chdir ~ptolemy/..; tar xf - )
   ```
   (this uncompresses the binaries and creates the executable files.) Note that is possible to install both the Sun-4 and the DecStation binaries on the same file system as different directories are used for each set of binaries.
4
   ```
   cd "~ptolemy"
   ```
   (change directories to the ptolemy root directory)
5. Edit ~ptolemy/.rhosts, adding a line for your system (This gives pigi permission to create an RPC server process with the *rsh* command). The line will be of the form "hostname ptolemy", you can find the correct hostname with the Unix command "hostname". Users will also need entries in their .rhosts files to allow the server process to start up.

1.3.2. Installation without creating a ‘‘ptolemy’’ user

The preferred installation technique, as indicated above, is to create a user called ‘‘ptolemy’’. The reason for this is that running Ptolemy requires an appropriate user configuration. At minimum, the user’s path must be set up properly. The ‘‘ptolemy’’ user is also configured to run an X window manager (twm) with suitable X resources that are known to work. In troubleshooting an installation, having the ‘‘ptolemy’’ user properly configured can be very valuable.

However, we recognize the some sites resist the creation of fictitious users. Consequently, Ptolemy can be installed anywhere. If you do this, *every* Ptolemy user must set a PTOLEMY environment variable to point to the root directory of Ptolemy. The installation is the same as above, except that ‘‘~ptolemy’’ is replaced with ‘‘$PTOLEMY’’.

1.3.3. Special considerations for installation under Open Windows

Ptolemy was developed using the X11R4 or X11R5 distributions from MIT. It will run successfully under Open Windows version 2. There are problems with running the graphic interface with Open Windows version 3, however. Some users have had no problems at all, but others have had intermittent problems such as "bad match" errors. We believe this may be a problem with the X-server supplied with the Open Windows 3.0, but the error is elusive and we have not yet tracked it down.

In order for all utilities included with this distribution to work under Open Windows (either version), you must install the shared libraries for the Athena widgets (the freely redistributable widget set from the MIT X11 distribution), which are provided with this distribution under the ‘‘~ptolemy/athena.sun4’’ directory. To install them, become root and copy all files in that directory into ‘‘/usr/openwin/lib’’ (or, if you have installed Open Windows in a non-standard place, into ‘‘$OPENWINHOME/lib’’). If you do not wish to do this, you could leave them in place and have every Ptolemy user change their LD_LIBRARY_PATH environment variable to search ‘‘˜ptolemy/athena.sun4’’ before ‘‘/usr/openwin/lib’’. Consult the Unix manual entry for the ‘‘ld’’ program to learn more about this.

After installation, the ptolemy account will contain several scripts for starting up X11R5 (Xrun), Open Windows with *olwm* (Xrun.ow), or Open Windows with *twm* (Xrun.ow.twm).

1.3.4. GNU Installation

For convenience, some software from the Free Software Foundation is included on the distribution tape. This includes the compiler used to make **Ptolemy**, which is also the compiler you will use to extend **Ptolemy**. Note that this is not a complete set of Gnu software.

To install the ptolemy gnu subset, proceed as follows:

1. `cd /usr/tools`
2. Check to see if there is already a "gnu" directory. If so, you probably don’t want to load the ptoelmy gnu files here and should pick a different location for the ptolemy gnu subset (be especially careful in step 6 below). If there is not already a "gnu" directory, then do the following command:
   ```
   mkdir gnu
   ```
3. If you have **Ptolemy** on tape, then do the following:

a. `cd /tmp` (or any directory where you have write permission and there is plenty, many Mbytes, of free disk space).

b. If you just installed Ptolemy from tape, and the tape is still in the tape drive you may skip the next step (c) and go to step (d), since you will be pointing to the second tar archive on the tape already.

c. insert the tape into the tape drive.

   ```
   mt -f /dev/nrst8 rewind
   ```
   (This rewinds the tape -- change the device name if your tape drive has a different name.)

   ```
   mt -f /dev/nrst8 fsf 1
   ```

d. `tar xf /dev/rst8`

4. If you have used FTP to get the ptolemy gnu file, then cd to the directory that contains the "tar.Z" file you downloaded via FTP.
5. You should now have a gnu-xxx.tar.Z file (where xxx is an architecture supported by ptolemy such as "sun4" or "mips") in your current directory.
6. Note, if they was already a "gnu" directory in "/usr/tools" (see step (2) above) you will need to change the commands below. Replace "/usr/tools" with the name of the parent directory of the location you have picked for the ptolemy gnu subset.

   If you are running on a Sun 4 (Sparc, IPC, IPX, etc) then:
   ```
   zcat gnu-sun4.tar.Z | ( chdir /usr/tools; tar xf - )
   ```
   If you are running on a DecStation (MIPS) then:

It is also possible to build **Ptolemy** with Sun’s port of AT&T’s *cfront* compiler, version 2.1. To do so, you must completely rebuild Ptolemy. The libraries on the tape were produced by the Gnu C++ compiler and are not interoperable with *cfront* code.

```
zcat gnu-mips.tar.Z | ( chdir /usr/tools; tar xf - )
```

The gnu tar files also contain the gnu source code in the /usr/tools/gnu/src directory. You will save disk space and **Ptolemy** will still run fine if you delete all of the gnu source files. However, if you plan to redistribute the tools (give them to anyone else) you must include sources, according to the Gnu Public License. Therefore, it may be a good idea to keep these source files around.

1.3.5. Testing the Installation

Note that the following tests assume that you have created a user "ptolemy" and installed the system there. One advantage of such an installation, is that the user "ptolemy" already has a working .cshrc and .login file to make startup easier.

To test **Ptolemy**:

1. login as ptolemy
2. If the X server is not already running, the .login script will attempt to start it. If your installation is different from ours, you may need to modify .login to work at your site (in particular, you may need a different *path* variable).
3. cd demo
4. pigi
5. Follow instructions in section 2 of the Almagest titled, ‘‘Running the Ptolemy Demos’’ (section 2.3.1).

Note that the ptolemy user provides a model of a user properly configured to run ptolemy. All the .files in the home directory are set up according to the tastes of the **Ptolemy** authors, and according the standard use of windowing software in the **Ptolemy** development group.

1.3.6. Rebuilding Ptolemy From Source

If you wish to rebuild **Ptolemy** from source (this step is recommended if you plan to do major development work, such as adding a new domain), it is simply a matter of editing the appropriate configuration file and typing ‘‘make’’. This is explained in a bit more detail below.

Configuration files live in the root directory of your ptolemy system. If you have installed PT as the user "ptolemy" then these file are in the ˜ptolemy home directory. The configuration file are all named ‘‘config-xxx.mk’’ where the ‘‘xxx’’ is something like "sun4" for a Sun-4 system or "mips" for a DecStation. They are included by other makefiles and define symbols specifying compiler flags, the directory where X include files are located, etc. The file that is used is determined by replacing ‘‘xxx’’ above by the value of the ARCH environment variable.

If you wish to rebuild using cfront rather than g++, use config-cfront.mk as a starting point to produce your configuration file. This has been tested with cfront 2.1 on a Sun-4; for other platforms, you may need to do some tweaking.

To rebuild the system, first adjust the configuration parameters in the appropriate configuration file. For example, if you are using the Gnu tools on a Sun-4, then you will need to adjust the "config-sun4.mk" file.

Next, run make. We recommend using Gnu make, which is included in the gnu subset. (Make sure that the Gnu tools are installed correctly.) Ultrix make will definitely not work.

You will get some warnings from the compiler, but the following warnings can safely be ignored:

any warning about file_id' defined but not used.

../../src/kernel/Tokenizer.cc:85: warning: cast discards 'const' from pointer target type

../../../../src/domains/thor/kernel/fprval.cc:56: warning: ordered comparison between signed and unsigned

../../src/ptcl/PTcl.cc:<n>: warning: cast discards 'const' from pointer target type (this warning may appear four times with different values of <n>)

../../src/ptcl/ptclMain.cc:75: warning: unused variable 'ptcl'

1.4. Trouble Shooting

This section is a collection common difficulties encountered when ptolemy is installed. This list is, of course, by no means complete. If you can't find your particular problem here, refer to "Additional Resources" at the end of this section.

1.4.1. Permission Denied

If when you start pigi you get the message "permission denied" and the **Ptolemy** greeting window fails to pop up, then Vem was unable to start pigiRpc, the associated **Ptolemy** process. This process is started using the Unix "rsh" command. This command can fail if your workstation does not have networking enabled, or the user's .rhosts file does not include the name of the workstation on which you are running. The following simple test checks your networking and .rhosts configuration:

```
rsh name echo "hello world"
```

where "name" is the name of your workstation. If you get back "hello world", then your workstation networking is correct. If you get the "permission denied" message again, then either your .rhosts is incorrect or networking is turned off on your workstation. If the latter is the problem, please ask your system administrator to enable networking.

1.4.2. Graphs fail to appear

If you succeed in invoking pigi, but graphs do not appear, but messages saying that the xgraph program could not be found appear instead, your path is set incorrectly. Because pigi uses *rsh* to start up **Ptolemy**, your path must be defined in your .cshrc file, not just in your .login file. The path must be defined even when ''prompt'' is not set, since your .cshrc file will be read in non-interactive mode.

1.4.3. Ptolemy won't recompile

You may be using a substantially different version of the Gnu compiler. The system is most likely to build if you use the same tools that we used originally. The gnu tools we used are supplied with the Ptolemy distribution.

1.4.4. Old Flowgraphs won’t work

Several stars that were included in the 0.3.1 release of Ptolemy have been renamed. The objective was to adopt a consistent naming policy across all domains. One major benefit of this is the ability to migrate universes and galaxies from one domain to another. A consequence, however, is that some of your universes and galaxies will have inconsistencies (they will point to icons that no longer exist in the current release). Please see the file "$PTOLEMY/lib/rename/README." This file explains how to upgrade your universes and galaxies to the current release.

1.4.5. Error: ld.so: libXext.so.4: not found

You have not installed the shared library needed by Ptolemy when it is used under Open Windows. See the "Open Windows" section in the installation guide.

1.4.6. Additional Resources

The best, most complete source for information on Ptolemy is to be found in the PT manual, the Almagest. Chances are, the information you are looking for is in there somewhere.

The entire manual is included in every distribution. In addition, the printed documentation is also available from the ILP office at the following address:

EECS/ERL Industrial Liaison Program Office
Software Distribution
479 Cory Hall
University of California, Berkeley
Berkeley, CA 94720
(510) 643-6687
email: software@hera.berkeley.edu

The current price is $25. US, but that is subject to change. Contact them for the exact fee.

A second source is the "ptolemy-hackers@ohm.berkeley.edu" mailing list. This list provides a forum for ptolemy questions and issues. Users of the current release who have a Ptolemy question, comment, or think they’ve found a bug should send mail to ptolemy-hackers.

To join ptolemy-hackers, send mail to
"ptolemy-hackers-request@ohm.berkeley.edu
with the Subject "ADD ME."

Another source is the Ptolemy ftp archive. To access this, use anonymous FTP to ptolemy.berkeley.edu.

1.5. Acknowledgements

Ptolemy has had a number of contributors. The overall coordinators are Prof. Edward A. Lee and Prof. David G. Messerschmitt of the EECS department at UC Berkeley, although there has also been involvement by the groups of Profs. Rabaey, Brodersen, Linnartz, and Kahn. Joseph Buck has been responsible for key management of the development of the kernel, and hence has impacted every aspect of Ptolemy. He also coordinated virtually all contributions, and authored both the interpreter and the ptlang preprocessor. He also designed the memory allocation system used by assembly language code generation domains. Other key contributors to the kernel include Soonhoi Ha and Ichiro Kuroda. Soonhoi Ha authored the DDF, DE, and CGC domains, including many of the basic stars and the basic domain interface, and also made extensive contributions to the CG domain, the kernel, and parallel schedulers of all types. Philip Bitar had a major impact on the design of the DE domain. José Pino has been primarily responsible for the code generation domains, although extensive contributions have been made by Joe Buck, Soonhoi Ha, Praveen Murthy, Thomas M. Parks, and Kennard White. Chih-Tsung Huang, with some help from José Pino, singlehandedly ported all the assembly code generation stars from Gabriel. Shuvra Bhattacharyya and Joe Buck wrote the loop scheduling mechanism, and Shuvra Bhattacharyya contributed the Gantt chart display tool. Seungjun Lee wrote the Thor domain, although some contributions to it have been made by Asawaree Kalavade. Ichiro Kuroda from NEC contributed to the development of the state handling mechanism. The graphical interface was written by Edwin Goei, and has been enhanced by several people, including Wan-Teh Chang. Andrea Casotto has provided modifications and improvements to VEM that have considerably improved the user interface.

Several people have had a major impact on the development of Ptolemy through their major efforts on its predecessor, Gabriel. Phil Lapsley has had incalculable impact on the directory structure, project management, documentation, and code generation efforts in Ptolemy. The first version of the graphical interface was written by Holly Heine.

Many people have had an impact on the current release by contributing stars and/or demo programs. These include, in addition to all the people mentioned above, Rachel Bowers, Rolando Diesta, Alireza Khazeni, Erick Hamilton, Gregory Walter, and Chris Yu (from the Naval Research Laboratories). Others had an indirect impact by contributing stars or demo programs to the predecessor program, Gabriel. These include Jeff Bier, Martha Fratt, Eric Guntvedt, Mike Grimwood, Wai-Hung Ho, Steve How, Jonathan Lee, Brian Mountford, Maureen O’Reilly, Andria Wong, and Anthony Wong.

Ptolemy is very much an ongoing project, with current efforts expected to be included in future releases. Participants in those efforts will be acknowledged when their work is included in a release.

1.6. SUPPORT

The Ptolemy project started in late 1989. A variety of sponsors have directly or indirectly supported the effort. Most notably, these include: AT&T, Bell Northern Research (BNR), Comdisco Systems, DARPA, Dolby Laboratories, GTE Government Systems, Hitachi America, Hitachi Ltd., Hughes Network Systems, Hughes Research Laboratories, Motorola, the National Science Foundation (NSF), NEC, the Office of Naval Technology (ONT) through the Naval Research Laboratories (NRL), Philips, Rockwell, the Semiconductor Research Corporation (SRC), the State of California MICRO Program, Sony, and Star Semiconductor. In kind contributions have also been received from Ariel, Berkeley Camera Engineering, Texas Instruments, Spectrum Signal Processing, Signal Technology Inc. (STI), and Xilinx. Other sponsors have contributed indirectly by supporting Gabriel, the predecessor.

1.7. References

[Bha92]

S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," to appear in *J. of VLSI Signal Processing*.

[Bie90]

J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E.A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro Magazine*, October 1990, Vol. 10, No. 5, pp. 28-45.

[Buc91]

J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy", *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, Toronto, Canada, April, 1991.

[Buc92a]

J. Buck and E. A. Lee, "The Token Flow Model," presented at Data Flow Workshop, Hamilton Island, Australia, May, 1992.

[Buc92b]

J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", invited paper, to appear in *International Journal of Computer Simulation*, special issue on "Simulation Software Development," 1992.

[Ha91]

Soonhoi Ha and E.A. Lee, "Compile-Time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration," *IEEE Transactions on Computers*, November, 1991.

[Ha92]

S. Ha, "Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs," Ph.D. Dissertation, EECS Dept., University of California, Berkeley, CA 94720, April 1992.

[Har86]

D. S. Harrison, P. Moore, R. Spickelmier, A. R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment," *Proc. of the IEEE Int. Conf. on Computer-Aided Design*, November 1986.

[Ho88]

W.-H. Ho, E. A. Lee, and D. G. Messerschmitt, "High Level Data Flow Programming for Digital Signal Processing", in *VLSI Signal Processing III*, Ed. R. W. Brodersen and H. S. Moscovitz, IEEE Press, New York, 1988.

[Kal91]

A. Kalavade, "Hardware/Software Codesign Using Ptolemy", MS Report, Electronics Research Laboratory, University of California, Berkeley, CA 94720, December, 1991.

[Kal92]

A. Kalavade, E.A. Lee, "Hardware/Software Co-Design Using Ptolemy - A Case Study ", to appear in *Proceedings of the IFIP International Workshop on Hardware/Software Co-Design*, Grassau, Germany, May 19-21, 1992.

[Lap91]

P. D. Lapsley, "Host Interface and Debugging of Dataflow DSP Systems", MS Thesis, Electronics Research Laboratory, University of California, Berkeley, CA 94720, December, 1991.

[Lee87a]
E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, January, 1987.

[Lee87b]
E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow" *IEEE Proceedings*, September, 1987.

[Lee88a]
E. A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages", in *VLSI Signal Processing III*, Ed. R. W. Brodersen and H. S. Moscovitz, IEEE Press, New York, 1988.

[Lee89b]
E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP", *IEEE Trans. on ASSP*, November, 1989.

[Lee91a]
E. A. Lee, "Consistency in Dataflow Graphs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, April 1991.

[Lee91b]
E. A. Lee, "Static Scheduling of Data-Flow Programs for DSP," in *Advanced Topics in Data-Flow Computing*, ed. J.-L. Gaudiot and L. Bic, Prentice-Hall, 1991.

[Lee91c]
E. A. Lee and J. C. Bier, "Architectures for Statically Scheduled Dataflow", reprinted in *Parallel Algorithms and Architectures for DSP Applications*, ed. M. A. Bayoumi, Kluwer Academic Pub., 1991.

[Lee92a]
E. A. Lee, "Data Parallelism in Graphical Signal Flow Representations of Algorithms", Technical Report, EECS Dept., UC Berkeley, August 13, 1992.

[Lee92b]
E. A. Lee, "A Design Lab for Statistical Signal Processing," *Proceedings of ICASSP*, San Francisco, March, 1992.

[Lee93]
E. A. Lee, "Multidimensional Streams Rooted in Dataflow", EECS Dept., UC Berkeley, August 1, 1992. to appear in *Proc. IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium-Grain Parallelism*, Orlando, FL, January, 1993.

[Mes84a]
D. G. Messerschmitt, "A Tool for Structured Functional Simulation", *IEEE Journal on Selected Areas in Communications*, SAC-2(1), January, 1984.

[Mes84b]
D. G. Messerschmitt, "Structured Interconnection of Signal Processing Programs," *Proc. of Globecom*, Atlanta, Georgia, 1984.

[Ous90]
J. Ousterhout, "Tcl: An Embeddable Command Language," USENIX Conference Proceedings, Winter, 1990.

[Ous91]
J. Ousterhout, "An X11 Toolkit Bassed on the Tcl Language," USENIX Conference Proceedings, Winter, 1991.

[Pin92]
J. Pino, S. Ha, E. Lee, J. Buck, "Software Synthesis for DSP Using Ptolemy", invited paper in the *Journal on VLSI Signal Processing*, special issue on "Synthesis for DSP", to appear.

[Sih92a]
G. C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", to appear, *IEEE Trans. on Parallel and Distributed Systems*.

[Sih92b]
G. C. Sih and E. A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," to appear in *IEEE Trans. on Parallel and Distributed Systems*.

[Tho86]
Thor Tutorial, VLSI CAD Group, Stanford University, 1986.

[Won92]
Anthony Wong, "A Library of DSP Blocks and Applications for the Motorola DSP96000 Family", Master’s Report, Plan II, EECS Dept., UC Berkeley, CA 94720, May, 1992.

2. The Ptolemy Interactive Graphical Interface

Authors:

Joseph Buck
Edwin E. Goei
Alan Kamas
Phil Lapsley
Edward A. Lee
David G. Messerschmitt

2.1. Introduction

This section contains all the information you need to run **Ptolemy** using the graphical interface. If you are familiar with the X window system, and your account is appropriately set up to run **Ptolemy**, then you may wish to skip directly to section 5 and begin running **Ptolemy** immediately. You may need to occasionally refer to section 2 below to understand some of the arcane terminology used.

2.2. Terminology

The **Ptolemy** interactive graphical interface (pigi) is a design editor based on tools from the Berkeley CAD framework. Using pigi, **Ptolemy** applications are constructed graphically, by interconnecting icons. Hierarchy is used to manage complexity, to abstract subsystem designs, and to mix domains. This document explains version 0.4, which operates with **Ptolemy** version 0.4.

In **Ptolemy**, applications are constructed by connecting **`Block`**s. In pigi, a block is represented by an icon with *terminals*, corresponding to its **`Porthole`**s in **Ptolemy**. The application hierarchy can be traversed by looking inside an icon to see how it is constructed. Elementary blocks are called **`Star`**s. Looking inside a star reveals the C++ code that defines it (actually, most stars are written in a special preprocessor language). A **`Galaxy`** is a block that contains stars and other galaxies. From the outside, a galaxy looks like a star, and can be treated exactly like a star. But looking inside a galaxy reveals the block diagram that defines it. A galaxy has input and output ports (corresponding to terminals on its icon) that specify its external connections, and has parameters that are linked to parameters of its internal blocks. The outermost block, which does not have any external connections, is called a **`Universe.`** A universe encapsulates an entire application.

A **`Wormhole,`** like a galaxy, looks like a star from the outside, and contains a block diagram on the inside. Unlike a galaxy, however, the model of computation (or the **`Domain`**) is not the same on the inside as on the outside. In pigi, wormholes are almost indistinguishable from galaxies. They are created automatically when two different domains interact. Domains that are currently supported in pigi include synchronous dataflow (SDF), dynamic dataflow (DDF), discrete events (DE), circuit simulation (Thor), code generation (CG), code generation with dynamic dataflow (CGDDF), C code generation (CGC, derived from CG), and assembly code generation for the Motorola DSP56000 family (CG56, derived from CG) and the DSP96000 family (CG96, derived from CG). See the manual sections for these domains for an explanation of their operation. Code generation domains are distinguished not only by the model of computation, but also by the language synthesized. Currently, all code generation domains except CG-DDF use the synchronous dataflow (SDF) model of computation. Knowledgeable users can define new domains and link them into pigi.

A **`State`** is a data-structure associated with a block, used to remember data values from one invocation to the next. For example, the gain of an automatic gain control is a state. A state need not be dynamic; for instance, the gain of fixed amplifier is a state. A *parameter* is the initial value of a state. Pigi is responsible for storing parameter values.

Pigi is built on top of existing CAD tools that are part of the Berkeley CAD framework. A central component of this framework is **Oct**, which is a design manager. Oct keeps track of block connections, parameter values, hierarchy, and file structure. **VEM** is an interactive graphical editor for Oct. VEM provides one of many ways to examine and edit designs stored by Oct. This section gives enough information about VEM to use it with **Ptolemy** in simple ways. It is assumed that VEM 8.1, which is part of the octtools distribution version 5, is being used. More complete documentation is contained in the VEM section of the Almagest. A third component of this system, called **RPC**, is a link between applications and VEM.

In pigi, the **Ptolemy** kernel runs as an RPC application. Users edit designs using VEM, store their designs using Oct, and execute their application through the RPC link to the **Ptolemy** kernel. The **Ptolemy** kernel and VEM are separate Unix[1] processes, called *vem* and *pigiRpc*. These processes interact with another using ‘‘remote procedure calls’’, hence the ‘‘Rpc’’ suffix.

Oct objects are called facets. A *facet* is the fundamental unit that a user edits with VEM. As an analogy, we can think of a facet as a text file in a computer system and VEM as a text editor, such as *vi* or *emacs*. However, instead of calling system routines to access the data stored in a text file like *vi* does, VEM calls Oct routines to access the data stored in a facet. Thus, Oct manages all data accesses to facets. Facets may define a universe, a galaxy, or a wormhole. They also define the physical appearance and formal terminals of icons that represent stars, galaxies, universes, and wormholes.

A facet may also contain a *palette*, which is simply a collection of disconnected icons. Palettes are directories of stars, galaxies, universes, and wormholes in a library. Note that facet names, like file names in Unix, may not contain spaces.

Here is a quick summary of terms:

Ptolemy	The entire design environment.
pigi	A **Ptolemy** graphical interface.
Oct	The design manager and database.
VEM	A graphical editor for Oct, upon which pigi is built.
Rpc	The link between VEM and the **Ptolemy** kernel.
facet	A fundamental object in Oct that the user edits with VEM.
palette	A facet that contains a library of icons.

---

[1] Unix is a trademark of AT&T.

star	Lowest level block in **Ptolemy**.
galaxy	A block in **Ptolemy** that contains sub-blocks, i.e., other stars or galaxies. A galaxy also has at least one input or output port.
universe	The outermost block in **Ptolemy** representing a complete system that the user can execute.
domain	A model of computation, or a set of semantic rules defining the behavior of a network of blocks. In code generation, a domain also corresponds to single target language.
wormhole	A block that does not have the same domain on the outside as the inside.

2.3. Setup

This section explains how to configure a user to run pigi. If someone has done this for you already, you may skip this section. It includes enough information about the window system for the novice user to get started.

Usually, when **Ptolemy** is installed on your system, a fictitious user named 'ptolemy' is created. This user is configured according to the tastes of the authors of Ptolemy. To configure your own account to behave the same way, you can copy all the dot files (hidden files whose names begin with a period) from the directory ˜ptolemy into your home directory. You can see these files with the ''ls -a ˜ptolemy'' Unix command. Unless you are a new user, however, you will probably not want to overwrite features already defined in your own dot files. In this case, you will need to extract the features you desire from the dot files in ˜ptolemy and them to your own. To make this easy to do, we explain below the salient features of each of these files.

2.3.1. Hosts

Pigi runs as two separate processes, one of which is started using *rsh* (remote shell). In order for the second process to start properly, you may need to have a file in your home directory called ''.rhosts'' with the line

```
machine_name user_name
```

where these names depend on who you are and what your machine is called. If rsh is not granted permission, you will receive an uninformative message saying ''Permission denied.'' If you see this, it means you must fix your .rhosts file.

As an alternative, the system manager may have set up the file ''/etc/hosts.equiv'' to make all the workstations on your cluster ''equivalent''. If so, you need not bother with .rhosts.

2.3.2. Environment Variables

The most basic requirement is that **Ptolemy** executables be in your path. **Important:** If you are a csh user, this should be done in your .cshrc file, rather than your .login file, because RPC applications are started as separate processes, and new processes do not examine the .login file.

The ˜ptolemy/.cshrc file defines the path variable to include the appropriate bin directories to run **Ptolemy**. The pertinent lines read:

```
if (! $?ARCH) setenv ARCH ‘/bin/arch‘
set path = ( . ˜ptolemy/bin ˜ptolemy/bin.$ARCH ˜ptolemy/octtools/bin ... )
```

The first of these lines sets your environment variable **ARCH** to the name of the type of machine you are running (e.g. sun3, sun4, vax, or mips). The second puts the **Ptolemy** and Oct bin directories into your path. If you add these lines to a previously existing .cshrc file, be sure to put these lines at the end of the file, after any other directives that set the path variable.

If you plan to print out any **Ptolemy** documents, you will need to set the PRINTER environment variable. In ˜ptolemy/.cshrc the pertinent line reads:

```
setenv PRINTER lw
```

You should replace **lw** with whatever printer name you are using.

If, for some reason, you wish to use an alternate version of the *pigiRpc* process, you will need to set your PIGIRPC environment variable to the complete pathname of the executable file for the ‘‘alternate pigiRpc’’ you wish to use (advanced users use this feature when they want to, say, link in their own domains to Ptolemy).

2.3.3. If There Is No ‘ptolemy’ User

If **Ptolemy** has been installed without creating a ‘ptolemy’ user, then you must set the environment variable PTOLEMY to point to the main directory of the **Ptolemy** installation. If **Ptolemy** is installed in /usr/tools/ptolemy, for example, your .cshrc file should read:

```
setenv PTOLEMY /usr/tools/ptolemy
```

Whenever the documentation refers to ‘˜ptolemy’, you should use ‘$PTOLEMY’.

2.3.4. Running Pigi using Open Windows

Sun does not provide the Athena widgets in the Open Windows distribution; these are required by the VEM editor (the Athena widgets are a freely distributable toolkit for X11 developed at MIT). For this reason, we have provided a copy of the library on the tape, in the directory ‘‘˜ptolemy/athena.sun4’’. The simplest way to install them is to simply copy the files in this directory to the directory containing the Open Windows libraries, which is usually ‘‘/usr/openwin/lib’’ or ‘‘$OPENWINHOME/lib’’.

Pigi will run under Sun’s Open Windows Version 2 or 3, once the Athena widgets are installed. This document will not attempt to teach you how to use Open Windows; there are excellent tutorial documents available from Sun. We’ll assume as a starting point that you are configured to run Open Windows; *do not* attempt to use the .login and .cshrc in the ptolemy account in this case. First, make sure that your .cshrc file defines the ARCH environment variable as above. Now add the directories ‘‘˜ptolemy/bin’’ and ‘‘˜ptolemy/bin.$ARCH’’ to your path, as described above. You’ll need to have the ‘‘olwm’’ window manager run in the mode where focus follows the mouse (in this mode, your keystrokes go to whatever window your mouse points to); to do this, find the line in your .xinitrc file that invokes olwm; the option **-follow** to the olwm command. If, for some reason, you must run olwm in the click-to-focus mode, you must click the mouse in each window before pigi will accept keyboard commands in that window; it is possible but annoying to use pigi this way. An alternative is to add the following line to your ‘‘.Xdefaults’’ file:

```
OpenWindows.SetInput: followmouse
```

There is a fatal interaction between VEM and the Open Look Window Manager (olwm) in Version 3 of Open Windows. One workaround is to use the *twm* window manager from the X11R4 distribution instead; it is included on the distribution tape, along with its manual entry, which can be found in ‘‘˜ptolemy/doc/man/man1/twm.1’’. This problem was discovered too late to know whether the problem is in Open Windows or in VEM; we apologize for making you use an uglier window manager.

2.4. The X Window System

Pigi uses the X window system, version 11 release 5 (X11R5). It will also run correctly under X11R4 and under Sun Open Windows. This section provides just enough information about using X so that if you are not familiar with it you will nonetheless be able to get started. Complete documentation, however would be helpful. Complete guides to the X window system are available in many technical bookstores. If you are familiar with X, and your user account is properly configured to run X11R4 or X11R5, you may skip the rest of this section, proceeding directly to the **Ptolemy** demos. Pigi will automatically load the resources it needs for operation. If you wish to customize these resources, it may be useful to scan this section. Resources are explained below.

2.4.1. Starting X

If your user account has already been set up to run **Ptolemy**, you may skip this section. There are two principal ways that X may be configured:

- The X server may be running all the time. In this configuration, when no one is logged in a single window together with a background appears. You don’t have to worry about starting X. A daemon program called *xdm* is responsible for logging you in.
- The user may be responsible for starting X – in this case, when you log in you get a bare Sun (or VaxStation or DecStation) tube.

Our sample configuration attempts to support both to some degree. If *xdm* is running, it will use the supplied .xsession file to set up windows. If xdm is not running, and the user is on a bare Sun tube, the *xinit* program will fire up, starting the X server; the .xinitrc file is used in this case to configure windows. The supplied .xinitrc file reads:

```
xrdb $HOME/.Xresources
twm&
xterm -geometry 80x30+0-0 -fn 9x15 -name "bottom left" &
xclock -geometry 120x120-0+0 &
exec xterm -geometry +0+0 -fn 9x15 -name "login" -ls
```

The first line reads X resources from the file ˜/.Xresources. Then the window manager called ‘‘twm’’ is started. The next three lines open windows using the xterm and xclock commands. The .xsession file works pretty much the same way for xdm installations. The function of the mouse buttons is determined by the ˜ptolemy/.twmrc file. This file is reasonably easy to interpret, even without any X experience, so you may wish to examine it and customize it. If your installation uses the X window system in some way that differs from our default, then whoever installed **Ptolemy** should have modified the .login file above to reflect this.

2.4.2. Manipulating Windows

If you already know how to use the X Window system and you are using your own window manager configuration rather than the standard **Ptolemy** configuration, you may skip this section. Assuming you use the file ˜ptolemy/.twmrc without modification, rather than your own window manager, the basic window manipulations are explained below. Note that there are often several different ways to accomplish the same objective. Feel free to experiment with key and mouse button combinations in various parts of the screen. First, you must identify the ‘‘meta’’ key on your keyboard. It may be labeled in any of various ways, including ‘‘meta’’, ‘‘left’’, ‘‘right’’, ‘‘alt’’, a small diamond, or any of a number of other possibilities. It is usually close to the shift and control keys. Most window manipulations require that you hold down the meta key while depressing mouse buttons.

Iconifying windows.

Depressing the meta key, and the left mouse button in any window will iconify it. The window disappears, replaced by a symbol in the icon manager at the right of the screen. To get the window back, place the cursor in the appropriate slot of the icon manager, and click the left mouse button. An alternative iconifying mechanism is to click any mouse button on the icon symbol at the left of the window header.

Moving windows.

Holding the meta key and dragging with the middle mouse button will move a window. ‘‘Dragging’’ simply means to hold the mouse button while moving the mouse. Alternatively, you can drag the left button in the middle of the window header.

Resizing windows.

The meta key and right mouse button can be used to resize a window. Place the mouse cursor near a corner of a window, depress the meta key, and drag the right button. Alternatively, without the meta key, any button in the rightmost corner of the window header will resize the window.

Mnemonic.

The window header has an iconify icon at the left, a blue bar for moving the window in the middle, and a resize symbol at the right. Correspondingly, without going to the window header, but using the meta key, the left mouse button will iconify a window, the middle button will move it, and the right button will resize it. Hence, the window header can be used as a mnemonic to help remember which mouse button has which function when the meta key is depressed.

Pick up and stuff.

In a text window, without the meta key, the mouse may be used to grab text and put it somewhere else, either in the same window, or in some other window. This can be very useful to avoid copying long sections of text. The left mouse button, when depressed and dragged, highlights a region of text in a window, and stores the text. The right button can be used to modify the extent of the highlighted region. The highlighted text is "picked up", stored for future use. The middle button causes the highlighted text to be typed to whatever window has the mouse cursor.

New windows.

A useful command, defined in ˜ptolemy/.cshrc, is ‘‘term’’. Typing this in any window, followed by a carriage return, opens a new terminal window with a reasonable size and color. You may change the size and location in the usual way.

Removing windows.

Most windows, including all pigi windows, can be removed by typing control-D with the mouse cursor in the window.

Resources

A *resource* in X is a parameter that customizes the behavior of applications that run under X. For instance, resources determine the foreground and background color of your windows. Pigi requires that some resources be set. These resources are defined in the file ˜ptolemy/lib/pigiXRes, and are automatically merged with whatever other resources you may have defined. The merging occurs when you invoke the startup script ˜ptolemy/bin/pigi. In addition to these required resources, there are many optional resources. Some of these are set in ˜ptolemy/.Xresources.

If you have not used the X window system before, you will probably not have a file with the name .Xresources in your home directory, and can simply copy the one from ˜ptolemy. This file defines some basic resources. If you already have a file with this name, then you can probably use the one you have as is.

VEM can be extensively customized using resources. Please see the VEM section of the Almagest for instructions.

2.5. Running the Ptolemy Demos

Once X is running, you can get some experience by running the Ptolemy demos. Any user can do this, although average users are not permitted to change the demos. If you feel compelled to change a demo, you can copy it to your own directory by using ‘‘cp -r’’. See the section below on copying and moving Oct facets.

2.5.1. Executing Demos

To execute a demo, in any xterm window, change to the demo directory by typing:

```
cd ˜ptolemy/demo
```

or if there is no ˜ptolemy,

```
cd $PTOLEMY/demo
```

Start Ptolemy via the Ptolemy graphical interface:

```
pigi &
```

You should get three windows, a VEM console window at the upper left of your screen, a startup palette below that, and a message window indentifying the version of **Ptolemy**. If you do not get the message window, you probably have gotten a ‘‘permission denied’’ error message. Your .rhost file is not properly configured. If you do get all three windows, then you have started two processes: the graphical editor, *vem*, and a process named *pigiRpc* that contains the pigi code and the Ptolemy kernel. A *vem* console window will open in the upper left corner, and a palette of demo directories directly below it. Ptolemy (which runs in the pigiRpc process) will automatically start and open a greeting window. Click any mouse button in this window to proceed. The remaining VEM windows can be moved and resized.

The palette contains icons that themselves represent palettes. If you have a color monitor, the outline on the icons is purple, the color used for icons representing palettes. In the current version of **Ptolemy**, nine palettes are included, one for each of the domains. We will begin with the palette at the upper left, which contains applications in the synchronous dataflow (SDF) domain.

Place the mouse cursor in the palette window, and click the middle mouse button. The VEM command menu appears. This menu is of a style called a ‘‘walking menu’’. The names displayed are only headers. To see the commands, you must move the mouse to the arrows at the right of the menu. A sub menu should appear. Clicking any mouse button with a command highlighted will execute that command. To remove the menu without executing any command, simply click a mouse button anywhere outside the menu.

The VEM menu is used for manipulating the graphical description of an application. We will explain these manipulations later. For now, you want to find and execute a demo. Place the mouse cursor on the icon labeled ‘‘SDF’’, at the upper left of the palette. Get the **Ptolemy** command menu by holding the shift key and clicking the middle mouse button. This menu has the same style as the VEM menu. Go to the ‘‘Window’’ sub menu, and execute the *look-inside* command. A new palette will open, containing icons representing further palettes. Look inside the first of these, labeled ‘‘Basic’’. The icons inside contain **Ptolemy** universes, or application programs.

Note in the **Ptolemy** menu that the *look-inside* directive has an ‘‘i’’ next to it. This is a ‘‘single key accelerator’’. Without using the walking menu, you can look inside any icon by simply placing the mouse cursor and hitting the ‘‘i’’ key on the keyboard. The single-key accelerators are extremely useful. In time, you will find that you use the menu only for commands that have no accelerator.

Look inside the first demo, labeled ‘‘butterfly’’, at the upper left. You will see a schematic. Try looking inside any of the icons in this schematic. This time, instead of graphics, you will see text that defines the functionality of the block. The syntax of this text is explained later in the Almagest. To run the butterfly system using the walking menu, place the mouse cursor anywhere in the window containing the butterfly schematic. Again holding the shift key, click the middle mouse button. Go to the ‘‘Exec’’ sub-menu, and select ‘‘run’’ by clicking any button. Notice that typing an ‘‘R’’ would have had the same effect. When asked for the number of iterations, click on ‘‘OK’’ (or hit ‘‘meta-return’’) to accept the default. When the run is finished, a graph appears. Try resizing and moving this displays. Experiment in this xgraph window with drawing boxes; to draw a box, just drag any mouse button. This causes a new window to open with a display of only the area that your box enclosed. Although the new window covers the old, if you move it out of the way, you can see both at once. Any of the now numerous open windows can be closed with a control-d.

A few additional window manipulations will prove useful almost immediately. In any of the VEM windows, you can closely examine any part of the window by drawing a box enclosing the area of interest and typing an ‘‘o’’. Like in an xgraph window, this causes a new window to open, showing only the enclosed area. Unlike xgraph windows, typing the ‘‘o’’ is necessary. In addition, you can enlarge a window using the standard X manipulation, and type an ‘‘f’’ to fill the window with the schematic. You can also zoom-in (or magnify) by typing a ‘‘z’’, and zoom-out by typing a ‘‘Z’’. These and other VEM commands are referenced again later, and documented completely in the VEM section of the Almagest.

You may wish to understand exactly how this butterfly example works. There are several clues to the functionality of the stars. After a while, the icons themselves will be all you will need. At this point, you can get several levels of detail about them. First, you will want to know the name of each star. If you closed the butterfly window, open it again. Try making the butterfly window larger (dragging the middle button while holding the meta key, if you are using the **Ptolemy** standard window manager), and filling the larger window with the fill (‘‘f’’) command. If you make it large enough, names will appear on each of the icons. You could have gotten the names as well by zooming. Find the **`Ramp`** star; it is the left-most star, with an orange triangle in the icon. To understand its function, place the mouse cursor on it, and execute the Other:profile command. Here ‘‘Other’’ refers to the command category and ‘‘profile’’ to the command in the submenu (you may also type ‘‘,’’). This command summarizes the behavior of a star. Further information can be obtained with the Other:man (‘‘M’’) command, however as of this writing this has not been fully implemented. To see the parameter values of the star, execute the Edit:edit-params command, which has accelerator ‘‘e’’. Notice that the *step* parameter is given as an expression, ‘‘PI/100’’. We will see later that a reasonably flexible expression language is supported to specify parameter values. The ultimate documentation for any star, of course, is its source code. You can look at the source code by looking inside the star.

You can safely explore other demos in the palette by the same mechanism. The demos this palette are documented in the SDF section of the Almagest, the Ptolemy manual.

2.5.2. A Quick Introduction to Targets and Code Generation

With your mouse cursor in the ‘‘butterfly’’ schematic window, execute the Edit:edit-target command (or just type ‘‘T’’). You will get a list of the available **`Target`**s in the SDF domain. The ‘‘default-SDF’’ target is selected. Change it to ‘‘compile-SDF’’. A dialog box appears with the parameters of the target. Click ‘‘Ok’’. Now run the system again (‘‘R’’). This time, instead of just running a simulation, **Ptolemy** will generate a standalone C++ program that implements the system. The program will be put in a directory off your home directory called ‘‘PTOLEMY_SYSTEMS’’. You can change the name of that directory by setting the appropriate Target parameter. **Ptolemy** will compile and execute the program for you. It will take a while, but eventually you will get the same plot that you got originally by running a simulation, assuming that the appropriate compiler (g++) is in your path, and that you are using the same version of the compiler that was used to build the **Ptolemy** libraries.

In a shell window, go look in the directory ˜/PTOLEMY_SYSTEMS. You will see the following files:

```
butterfly
butterfly.cc
code.cc
make.template
makefile
```

The first of these is executable. Try executing it. You can modify the number of sample points generated using a command-line argument. For example, to generate 1,000 points instead of 10,000, type

```
butterfly 1000
```

You have just seen an example of an alternative **`Target`** within the SDF domain. A **`Target`** defines the way a system is executed. The default target is essentially an interpreter. The compile-SDF target synthesizes and standalone program and then executes it on the native workstation. More elaborate targets parallelize the code and execute the resulting programs on remote hardware. Targets can be defined by users and can make use of existing **Ptolemy** schedulers. Knowledgeable users can also define their own schedulers.

The resulting C++ program is fairly large because many **Ptolemy** objects get linked in (as of this writing, many *unnecessary* objects get linked in, so the executable is unnecessarily large). You can create a much leaner program by switching to the CGC domain.

2.5.3. A Quick Introduction to Domains

In **Ptolemy**, a **`Domain`** has two aspects, a ‘‘model of computation’’ and a ‘‘target language’’. The model of computation is the *operational semantics* of the network of blocks. It defines what is meant by an interconnection of blocks, and how the interconnection will behave when executed.

In the SDF domain, where you have been working, the model of computation is ‘‘synchronous dataflow’’ and the target language is C++. For an explanation of SDF semantics, see the SDF section of the Almagest. Many domains have been made for Ptolemy. SDF is used as an example only because is it is the most developed of all. The DDF (dynamic dataflow) domain is a superset of the SDF domain. It uses a run-time scheduler instead of a compile-time scheduler (see the DDF section of the Almagest). All SDF stars can be used in the DDF domain, but there is also a library of DDF-specific stars that require run-time scheduling. Since the butterfly example is built using only SDF stars, you can it using the DDF scheduler by just switching domains. Execute the Edit:edit-domain command (‘‘d’’), and select ‘‘DDF’’. You must also execute Edit:edit-target (‘‘T’’) to avoid trying to run the DDF scheduler with an SDF target. Then run the system (‘‘R’’). It will behave very much as it did under SDF, except that it might be slightly slower.

Another domain, called CGC or ‘‘C Code Generation’’, also uses synchronous dataflow semantics, but has C as its target language. Unlike DDF, this domain is not a superset of SDF. Not all SDF stars can be used in the CGC domain because it has a separate star library. However, we have made every effort use consistent naming of stars and their parameters so that it is easy to move from one domain to another. The ‘‘butterfly’’ example can be executed under the CGC domain. To do this, edit the domain (‘‘d’’) and select ‘‘CGC’’. Edit the target (‘‘T’’) and select default-CGC. Run the system (‘‘R’’). A window will open displaying a standalone C program that implements the system. Feel free to examine it. The program is put (by default) in ˜/PTOLEMY_SYSTEMS, and compiled. Assuming the compile and link options (Target parameters) are set correctly, and the compiler is in your path, then you will get an executable, again named ‘‘butterfly’’. The program is considerable smaller and somewhat faster than the standalone C++ program. As of this writing, it does not accept command-line options, although this is likely to change in the future.

2.5.4. Dialog Boxes

As you explore the demos, you will frequently encounter VEM dialog boxes. For example, the run command triggers a box asking for the number of iterations. To use the default number, you need to click ‘‘OK’’ or type the key combination indicated in the ‘‘OK’’ button (meta-return). To change the number, you must use EMACS-like editing control characters. These are summarized below:

Dialog Box Editing Control Characters

Key	Description
Delete, control-h	Delete previous character.
control-a	Move to beginning of line.
control-b	Move backward one character.
control-d	Delete next character.
control-e	Move to end of line.
control-f	Move forward one character.
control-k	Kill (delete) to end of line.
meta-b	Move backward one word.
meta-f	Move forward one word.
meta-h, meta-delete	Delete previous word.

In addition, using the mouse you can position the cursor anywhere in the text to begin editing by clicking the left button. The left and right arrow keys may also be used to move the cursor.

For example, to enter a new number for the number of iterations to run, position the cursor in the number box, type control-k followed by the new number. Do not type return; this adds a second line to the dialog entry, which for most commands is confusing at best. If you accidentally type return, you can backspace sufficiently to get back to one line.

Many dialog boxes have more than one field. The current field is highlighted, and anything you type will go into it. To change the current field to a different one, click the left mouse button on the new one, or use the ‘‘Tab’’ key to move to the next one.

Since you started pigi in a directory in which you do not have write permission, it is not possible here to create new universes, nor to modify the demos. The easiest way to proceed (when you are just learning) is to exit pigi before proceeding to the next section, and return to a directory where you have write permission. Do this by closing all xgraph windows that you have open by typing ‘‘control-d’’, and then typing ‘‘control-d’’ in the VEM console window. A dialog box may appear with a menu of facets that VEM thinks have been changed. Since all of these belong to the user ‘‘ptolemy’’, you do not want to save them. Do not select any of them. Just click ‘‘OK’’. Then a warning window will appear telling you that closing the console window will terminate the program. Just click ‘‘Yes’’.

2.6. Creating Your Own Universes

In this section, we will show how to create your own universes with a simple example. First, be sure you are in a directory where you have write permission, like your home directory.

- Create a new work area:

```
mkdir example
cd example
```

- Start pigi:

```
pigi
```

You will see the message:

```
creating initial facet ‘‘init.pal’’
```

Wait until the startup window with the version number appears. We are now ready to learn about the basics of using VEM. More complete VEM documentation can be found in the VEM section of the Almagest.

2.6.1. Terminology

As we said earlier, the fundamental Oct objects that we edit with VEM are called *facets*. This means that we read an entire facet from disk into VEM, make changes to it, and then save the facet back to disk.

Facets are specified by three names separated by colons. This is usually written as ‘‘cell:view:facet’’. The first component is the cell name which is used to name design. In pigi, the second component, called the view name, will always be ‘‘schematic’’.[2] Last, there is the facet component which can either be ‘‘contents’’ or ‘‘interface’’. The former is used for block diagrams, and the latter for icons. Note that all three facet component names may not contain any spaces.

Note also that the term ‘‘facet’’ can refer to two different things. It can mean the Oct object called a facet, or it can refer to the name of the last component of a facet’s name. The intended meaning is usually clear from context. If the facet component is not specified, VEM assumes that you mean the contents facet. Thus, ‘‘wave:schematic’’ refers to the facet with cell name ‘‘wave’’, view name ‘‘schematic’’, and facet name ‘‘contents’’.

The two kinds of facets, contents and interface, are used to store two different kinds of data. Contents facets contain the actual definition of a design while interface facets contain schematic symbols. In our example, we will create a universe with three stars. The universe will be stored

---

[2] Other Oct applications use other views such as ‘‘symbolic’’ or ‘‘physical’’ but these are not used by pigi.

in a contents facet. The three stars in our system will be represented by schematic symbols called **icons**. These icons are stored in interface facets. The universe itself can be given an icon, which will become its interface facet.

2.6.2. Building an application

Now we are ready to create a simple universe. Let's create a simulation that generates a sine wave and displays it.

- Open a new facet: The facet that is already open, called ‘‘init.pal:schematic’’, is special because pigi always opens a facet by this name in the directory in which it starts. Convention in Ptolemy dictates that ‘‘init.pal’’ should be used to store icons representing complete applications, so instead of using this facet, we will create a new one.
  - Place cursor in window labeled ‘‘init.pal:schematic’’.
  - Select the *open-facet* command from the ‘‘Window’’ pigi menu (shift-middle-button). Alternatively, type an ‘‘F’’.
  - Replace the cell Name ‘‘untitled’’ in the dialog with ‘‘wave’’ and click the ‘‘OK’’ button. A quick way to delete the ‘‘untitled’’ is using control-k. Leave the View and Facet fields unmodified. A new blank window will appear.
- Open a palette:
  - Place cursor in either blank window.
  - Select the *open-palette* command from the ‘‘Window’’ pigi menu. Alternatively, type an ‘‘O’’.
  - Pigi will present a palette menu. Select the ‘‘sdf’’ palette by clicking the left button in the box next to ‘‘$PTOLEMY/src/domains/sdf/icons/main.pal’’ (the first entry), and then click on ‘‘OK’’. This palette shows the basic categories of synchronous dataflow stars that are availabel. There are too many stars to put in just one palette. You can use the Window:look-inside (‘‘i’’) command to open any of the palettes. At this point you should look inside the ‘‘Signal Sources’’, ‘‘Nonlinear Functions’’, and ‘‘Signal Sinks’’. Arrange these palettes on the screen so that you can see the blank window labeled ‘‘wave:schematic’’.

2.6.3. Some Basic VEM Commands

At this time, it is worth exploring some basic VEM commands for manipulating window displays. VEM uses post-fix commands. This means that the user enters the arguments to a command before the command name itself. Arguments appear in the VEM console window as the user enters them.

There are several types of arguments. Each argument type is entered in a different way. All graphics arguments are created with the left mouse button. The five types of arguments are listed below:

- Point: position the cursor, click the left mouse button
- Box: position the cursor, drag[3] the left mouse button

---

[3] ‘‘Drag’’ means to press down on a mouse button, move the mouse while holding it down, and then release the button.

- Line: make a point, position the cursor above point, drag the left mouse button
- Object: use *select-objects* and *unselect-objects* commands (explained later)
- Text: enclose text in double quotes

Arguments can be removed from the command line by typing the delete key, backspace key, or ‘‘control-u’’, which deletes all the current arguments. There are three ways to enter commands:

- Menus: middle-button for VEM commands, shift-middle-button for pigi commands. Menus are of the ‘‘walking’’ variety, as explained before.
- Key bindings: commands can be bound to single keys and activated by just pressing the key. Key bindings are case sensitive. We have been giving key bindings for the pigi commands that have them as we go along. There are also tables at the end of this document summarizing VEM and pigi commands and their key bindings.
- Type-in: type a colon followed by the command name

Let’s try a few examples, some of which should be familiar by now. Place the cursor in the palette window (the one with the library stars) and:

- Type ‘‘shift-Z’’ (capital Z) for *zoom-out*. This makes everything smaller.
- Type ‘‘z’’ (lower-case z) for *zoom-in*. This makes everything bigger. If you zoom in sufficiently, labels will appear below each icon giving the name of the star. VEM does not display these labels if they will be too small.
- The VEM *show-property* command (‘‘control-p’’) is useful for finding out the name of a star when the label is not visible. Just place the mouse cursor on the icon of interest and type ‘‘control-p’’.
- Try ‘‘p’’ for *pan*. Pan moves the spot under the cursor to the center of the window.
- The VEM *pan* command can also take as an argument a point which will indicate the new center of the window. Recall that the argument must be entered first. Place a point somewhere in the palette window by clicking the left button, and type ‘‘p’’. The location of your point became the center of the window.
- The VEM *open-window* command can take as an argument a box. Draw a box in the palette window by dragging the left mouse button. Then type ‘‘o’’, or find the *open-window* command in the VEM menu.
- Try placing points in the new window. Notice that they also appear in the original palette window. Also notice that you are only permitted to place points at certain locations. VEM has an implicit *grid* to which points *snap*. The default snap resolution is suitable for making **Ptolemy** universes.
- You can get rid of your point (or any argument list) by typing ‘‘control-u’’. You can delete arguments one-at-a-time by typing ‘‘control-h’’. Try placing several points and then deleting them one by one.
- You can close the new window (or any window) with ‘‘control-d’’.
- A particularly useful command at this time is *show-all*, or ‘‘f’’. Try this command in the palette window. This rescales and recenters the display so that everything in the facet is visible.
- You can also resize a window, using whatever X Window bindings you have installed, and then type ‘‘f’’ to show all.

2.6.4. Back to the example

- Create an instance of the star called ‘‘Ramp’’. This star is the first star of the second row of the sources palette. Its icon has an orange triangle. To do this:
    - With your mouse cursor in the window ‘‘wave:schematic’’:
    - Create a point anywhere in the window by clicking the left button.
    - Move the cursor over the ‘‘Ramp’’ icon in the palette and press the ‘‘c’’ key. This is a key binding that executes the VEM ‘‘create’’ command.
    - You have just created an **instance** of the ‘‘Ramp’’ icon. The actual data which describes how the ‘‘Ramp’’ icon should be drawn is stored in another facet (an interface facet). An **instance** of the ‘‘Ramp’’ icon points to this facet.
- Delete and select instances: Sometimes in the process of editing your schematic, you may need to delete objects. As an example, let’s create another Ramp instance and then delete it.
    - Create another Ramp instance next to the first one: place a point near the original Ramp, place the cursor over the Ramp icon in the palette and press ‘‘c’’. Actually, you don’t have to use the icon in the palette – you could also put the cursor over the already existing Ramp icon to achieve the same effect.
    - Place the cursor over the new Ramp icon and execute *select-objects* by typing ‘‘s’’. This creates an object argument on the VEM command line. This is necessary because the VEM *delete-objects* command takes arguments of type ‘‘object’’. The *select-objects* command takes point, box, and/or line arguments and turns the items underneath them into object arguments. The *unselect-objects* command (‘‘u’’) does the reverse of *select-objects*.
    - Execute *delete-objects* by typing ‘‘D’’ (upper-case!). This deletes the objects we selected previously.
    - You could also have deleted the newly created Ramp with the *undo* command (‘‘U’’). This is an infinite undo, so you can backtrack through all changes you have made since starting the VEM session by repeatedly executing the undo
    - Occasionally when you use the select and unselect commands, the objects are not redrawn correctly. When this happens, use the VEM *redraw-window* command, ‘‘control-l’’ (lower case L), to redraw.
- Create the remaining instances in our example:
    - Create an instance of the ‘‘Sin’’ icon to the right of the Ramp. ‘‘Sin’’ is the first icon in the second row of the ‘‘nonlinear’’ palette. Make sure it does not overlap with the Ramp icon. If it overlaps, you can delete it and create a new one.
    - Create an ‘‘Xgraph’’ instance to the right of the Sin icon. ‘‘Xgraph’’ is the first icon in the first row of the ‘‘sinks’’ palette.

We now have three icons: a Ramp, a Sin, and an Xgraph. Next, we will connect them together.

- Connect the Ramp output to the sin input using the following steps:
    - With the mouse cursor in the ‘‘wave:schematic’’ window, type ‘‘f’’ to show all. This will rescale your system, and make it easier to make connections.

  - Draw a line between the output of the Ramp and the input of the Sin: put the cursor over the Ramp output, double-click on the left mouse button, and drag the cursor to the Sin input, and then let up on the mouse button. If the two terminals are not on a horizontal line, you can bend the line by momentarily releasing the mouse button while dragging it.
  - Type ‘‘c’’ for *create* to create a wire. Notice that the *create* command creates wires or instances depending on the type of arguments it is called with.
  - If you need to delete a wire, you can draw a box around it (click and drag with the mouse), select it (press ‘‘s’’), and then delete it (‘‘D’’).
- Connect the Sin output to the Xgraph input in a similar way.
- Run the universe: We now have a complete universe that we can simulate.
  - Execute the *run* command from the pigi ‘‘Exec’’ menu, or type an ‘‘R’’.
  - Enter ‘‘100’’ for the number of iterations. Do this by typing ‘‘control-k’’ to remove the default entry, typing 100, and clicking ‘‘OK’’. Instead of clicking ‘‘OK’’ you can also hit ‘‘meta-return’’.
  - A new window with a graph of a rough sine wave should appear. The execution of the Xgraph star created this new window which shows the output of our simulation. To remove this window, click on the ‘‘Close’’ button or press ‘‘control-d’’ in it.
- Save the facet by typing ‘‘S’’ (upper-case) with the cursor in the ‘‘wave:schematic’’ window. This executes the VEM *save-window* command. It is wise to periodically save your work in case the editor or computer system fails unexpectedly.
- Change parameters: If we look at the output, the sine wave appears jagged. This is because the Ramp star has a set of default parameters which cause the Ramp to generate output values with an increment that is too large. We can change the parameters of the Ramp star as follows:
  - Place the cursor over the Ramp icon and execute *edit-params* in the pigi menu (or type ‘‘e’’). A dialog box will appear that shows the current parameters of the Ramp.
  - Replace the value of ‘‘step’’ with ‘‘PI/50’’. (You can use ‘‘control-k’’ to erase the old value.) Finally, click the ‘‘OK’’ button to store the new parameters. This is an example of Ptolemy’s parameter expression syntax. This syntax supports addition, subtraction, multiplication, and division, written exactly as they would be in C or Fortran (exponentiation is not yet supported). The built-in symbol PI has the value 3.14... as you would expect.
- Run the simulation again using 100 iterations. This time the output should look like one cycle of a sine wave.
- Use *save-window* again to save the new parameters.
- To be able to conveniently access this example again, you should create an icon for it. To do this, use the pigi command ‘‘Extend:make-schem-icon’’, or ‘‘@’’. Just place the mouse cursor in the ‘‘wave’’ facet window, and hit the ‘‘@’’ key. A dialog box appears asking for the name of the palette in which you would like to put the icon. By convention, we put universe icons in palettes called ‘‘init.pal’’. So replace the default entry (which should be ‘‘./user.pal’’) with ‘‘init.pal’’. When making the icon is finished, find the ‘‘init.pal’’ window that first opened when you started the system, and type ‘‘f’’ to show all. An icon will

appear. Looking inside this icon (‘‘i’’) will get you your ‘‘wave’’ palette. The second item in the palette is a marker indicating where the next icon that you create will go. Henceforth, anytime you start pigi in this same directory, the first window you will see will be this ‘‘init.pal’’ window.

- Our example is now complete. To exit:
  - Close all xgraph windows with ‘‘control-d’’.
  - Type ‘‘control-d’’ in the VEM console window. A dialog box will appear asking you to choose buffers to be saved. Unfortunately, as of this writing, some of the buffers listed may have already been saved and do not need to be saved again. Also, some of the buffers listed may not have been changed, and therefore do not need to be saved. To indicate which of the listed buffers you wish to save, click on the box to the left of each name. Then click on the ‘‘OK’’ button.
  - A final warning appears telling you that closing the console window will terminate the program. Click on ‘‘Yes’’.

2.7. Using Galaxies

In this section we will explain how to create galaxies. Galaxies allow you to partition your design into more manageable pieces and to re-use designs as components in other designs.

2.7.1. Creating a Galaxy

Let’s use the schematic we created in the last example to make a sine wave generator galaxy.

- Instead of modifying our previous example, we will make a copy of it. In your ‘‘example’’ directory, type:

  ```
  cp -r wave singen
  ```

  The recursive copy, ‘‘cp -r’’, is necessary because Oct stores data using a hierarchical directory structure. Of course, if the facet singen exists already, you must remove it with ‘‘rm -r’’ first before copying.
- Start pigi.
- Use *open-palette* (or ‘‘O’’) to open the ‘‘$PTOLEMY/lib/colors/ptolemy/system’’ palette (the last one in the list of palettes). The system palette contains input and output ports which can be instantiated into schematics just like stars.
- Use *open-facet* (or ‘‘F’’) to open the ‘‘singen:schematic’’ you created using ‘‘cp -r’’. You must replace the default name ‘‘untitled’’ with ‘‘singen’’.
- In the ‘‘singen:schematic’’ window, delete the Xgraph star and the wire attached to it. The easiest way to do this is draw a box (click-drag) around the star and its input wire, press ‘‘s’’ to select these objects, and then press ‘‘D’’ to delete them. (You may want to enlarge your window to make it easier to work.)
- Place an output port where the Xgraph star used to be and connect it to the output of the Sin star. The output port is the icon in the system palette with an arrowhead (an input port, by contrast, has a fish tail).

- Name the output port ‘‘out’’:
  - Position the cursor over the black box on the new output port.
  - Type "out" as a text argument, including quotation marks.
  - Type ‘‘c’’ for *create*. Again, note that the *create* command has a different action than before. It names output terminals when given a text argument.
- We now have a galaxy. The fact that a schematic has input or output ports distinguishes it as a galaxy. This galaxy that you just created is similar to the ‘‘singen’’ galaxy in the ‘‘Signal Sources’’ palette. Find it, and look inside, to make the comparison.

2.7.2. Using a galaxy

We have just created a galaxy that we would like to use in another design. In order to do this, we need to create an icon for this galaxy which we will then instantiate in our other design.

- Create an icon:
  - Place cursor in ‘‘singen:schematic’’ window.
  - Execute *make-schem-icon* in the pigi ‘‘Extend’’ menu (‘‘@’’).
  - The dialog box should contain:

    ```
    Palette:  ./user.pal
    ```

    This specifies the name of the palette that will contain your icon. By convention, we usually put galaxy icons in the palette called ‘‘user.pal’’ in the current directory. Hence, this is the default name.
  - Click ‘‘OK’’. Since you had already created an icon for the ‘‘wave’’ universe, and that icon was copied by your ‘‘cp -r’’, VEM asks whether it is OK to overwrite the icon. Click ‘‘OK’’. Wait until *make-schem-icon* is done. VEM informs you that it is done with a message in the VEM console window, which may be buried by now. You should find it and bring it to the surface.
- Open the palette called ‘‘user.pal’’ using *open-palette* (‘‘O’’). The newly created galaxy icon should appear in this palette along with a bright yellow dot called a **cursor**. A **cursor** distinguishes a palette from other types of facets and it determines where the next icon will be placed.
- At this point, we have an icon for our sine wave generator galaxy. The icon for this galaxy is stored in an interface facet called ‘‘singen:schematic:interface’’. It is this icon facet which is instantiated in the ‘‘user’’ palette. We can now use our sine generator galaxy simply by instantiating our icon into another schematic.
- Use *open-facet* to create a new facet with the cell name ‘‘modulation’’.
- In the ‘‘modulation:schematic’’ window, create a universe which takes two ‘‘singen’’ galaxies, multiplies their outputs together, and then outputs the result using ‘‘Xgraph’’. The ‘‘Xgraph’’ star can be found again in the ‘‘Signal Sinks’’ subpalette of the SDF palette. The multiplier star, called Mpy, is in the ‘‘Arithmetic’’ subpalette. **Hint**: If you place the icons so that their terminals fall on top of one another, then a connection gets made without having to draw a wire.

- Now run the universe with the number of iterations set to 100. The output should appear as a squared sine wave, which is just an inverted sine wave of twice the frequency shifted up by 1/2.

So far, we have created a galaxy and used it in another universe. But, we could also have used our galaxy within another galaxy. In this way, large systems can be broken up into smaller more manageable pieces.

2.7.3. Galaxy and Universe Parameters

One of the problems with the ‘‘singen’’ galaxy that we just created is that it generates sine waves with a fixed frequency. We would like to make the frequency of the generator parameterizable. That way, we could set the two ‘‘singen’’ galaxies in our ‘‘modulation’’ universe to two different frequencies.

To make a galaxy parameterizable, we create **formal parameters** in the galaxy and then link the formal parameters to the **actual parameters** of the instances contained in the galaxy. The terms, formal and actual parameters, are analogous to formal and actual parameters in any procedural programming language. An example will make this clear.

- Create formal parameters:
  - Place the cursor in the ‘‘singen:schematic’’ window but away from any instance, i.e., in the grey background of the facet.
  - Execute *edit-params* (‘‘e’’) and a message will appear indicating that the Galaxy has no parameters. To add parameters to the galaxy, execute *define-params* (‘‘+’’) anywhere in the galaxy to get a dialog box with three blanks

```
New name:
New type:
New value:
```

Fill in the dialog as follows:

```
New name:  freq
New type:  float
New value:  PI/50
```

Then click on ‘‘OK’’. Recall that you can use ‘‘tab’’ to move from one field to the next of the dialog box and ‘‘meta-return’’ instead of ‘‘OK’’. Hence, the dialog can be managed from the keyboard, without requiring the mouse.

A common error in making entries in these dialog boxes is to type ‘‘return’’. This has the unfortunate consequence of making a multi-line entry, probably not desired in this case. Although you can correct the problem by typing repeated backspaces, it is usually easier to cancel the dialog and start over.

  - We just created a new formal parameter called ‘‘freq’’ with a default value of ‘‘PI/50’’. Additional parameters may be added or old ones changed simply by invoking *define-params* again. The value of a formal parameter can be changed by executing *edit-params* in the background of the galaxy, or on the icon representing the galaxy.

Exactly the same procedure can be used to attach formal parameters to a universe.

- Link formal parameters to actual parameters:
  - Place the cursor over the ramp icon in ‘‘singen:schematic’’.
  - Execute *edit-params* and fill in the dialog as follows:

    ```
    step:  freq
    value:  0.0
    ```

    This allows the ‘‘freq’’ parameter of a singen instance to control the increment of the internal ramp star, thus controlling the output frequency.
  - In earlier versions of pigi, it was necessary, when adding a formal parameter to a galaxy, to delete and recreate all instances of that galaxy. This is no longer required: all instances of the galaxy that already existed will immediately ‘‘know’’ about the new parameter, and will use the default value.
- Change the frequency of one of the ‘‘singen’’ instances to ‘‘PI/5’’ by using *edit-params*. This ‘‘singen’’ will be ten times the frequency of the other.
- Run the universe with an iteration of 100. The output should show the product of two sine waves with different frequencies. Don’t forget to save the facets we just created. It would also be a good idea to create an icon for ‘‘modulation’’ and put it in ‘‘init.pal’’.

2.7.4. Sounds

On some workstations (currently only Sun Sparcstations), **Ptolemy** can play sounds over the workstation speaker. You can try this with the universe you just created. Replace the ‘‘Xgraph’’ star in ‘‘waveform:schematic’’ with an instance of the ‘‘Play’’ star (second row of of the sinks palette, right of center, with a stylized loudspeaker as an icon). Edit the parameters of the ‘‘Play’’ star entering 16000 for the gain parameter. (To see details about the ‘‘Play’’ star, execute the ‘‘profile’’ command in the ‘‘Other’’ menu, or type a comma (‘‘,’’) with the mouse on the ‘‘Play’’ icon). The Sparcstation speaker is driven by a codec that operates at an 8kHz sample rate. So running this universe for 40000 samples will produce about 5 seconds of sound. The sound produced by the current parameters is not particularly attractive. Experiment with different parameter values. Try PI/1000 in place of the PI/50.

An interesting variant of this system modulates a chirp instead of a pure sinusoid with a low frequency sinusoid. A chirp is sinusoid that scans a frequency range. You could replace one of your ‘‘sinegen’’ instances with something that generates a chirp, and again experiment with parameters.

A chirp can be created with three stars: a ‘‘FloatRamp’’, an ‘‘Integrator’’ and a ‘‘Sin’’, connected in series. The *step* parameter of the ‘‘FloatRamp’’ should be very small, such as 0.0001. With this value, you will hear some aliasing if you create 5 seconds of sound. The ‘‘Integrator’’ is the seventh star in the fourth row of ‘‘main.pal’’, on the right, and its default parameter values are OK. Use the ‘‘profile’’ command (‘‘,’’) to read about it. Note that a fourth star, a ‘‘FloatDC’’ (upper left of ‘‘main.pal’’) is needed to set the ‘‘Integrator’’ *reset* input to zero.

This is a good time to try out your own examples. In general, when you create new galaxies and universes that depend on each other, it is a good idea to keep them together in one directory. For example, all of the facets we have created so far are in the ‘‘example’’ directory. You can use the extensive **Ptolemy** demos as models.

2.7.5. An Important Note

Whenever the contents of a galaxy are changed so that the new definition has different I/O ports, the icon must be updated as well. You can do this by calling *make-schem-icon* again to replace the old icon with a new one. VEM will not allow you to overwrite the icon if you have instances of the old icon in any open window (regardless of whether the window is iconified). Hence, you must either close those windows with ‘‘control-d’’ or delete the offending icon before replacing it with a new icon.

2.8. Hardcopy

The user has several options for printing graphs and schematics developed under **Ptolemy**. The first option generates a postscript description and routes it to a printer. The second uses the screen capture capability of the X window system.

2.8.1. Printing Oct Facets

A block diagram under pigi is stored as an Oct facet. To print it to a postscript printer, first save the facet. Do this by moving the mouse to the facet and press the "S" key to save. Then, keeping the mouse in the facet, invoke the "print facet" command from the "Other" pigi menu. You must have the environment variable PRINTER set to point to a valid postscript You can do this by putting the following line in your .cshrc file:

```
setenv PRINTER printername
```

You will have to restart pigi for this change to be registered.

Upon running the print facet command under pigi, a dialog box appears into which options may be entered. The following options may prove useful:

-p	Print in portrait mode, scaling to fit on the page or in the box defined by the -b option.
-l	Print in landscape mode, scaling to fit on the page or in the box defined by the -b option.
-a	Choose -l or -p, which ever gives the biggest plot (default).
-b	Give width and height to fit the plot in (in inches, defaults to the full page). For example, to get an image within a square five inches on a side, use "-b 5x5".
-s	Multiply the scale determined by -a, -l, or -p by ‘scale’ (defaults to 1.0).
-o	Additional offset (in inches).
-x	Print labels.
-X	Used with ‘-x’, only draw labels that fit in the bounding box of the label.
-A	Do not print anything that is smaller then ‘abs’ points (there are 72 points per inch)
-C	Generate code for color postscript printers.

-n Print the names of the terminals on icons.
-P Replace the standard PostScript prologue file with ‘prologue’.
-F Change the font and size (defaults to Times-Bold/7).

2.8.2. Capturing a Screen Image

Under the X window system and compatible systems there are facilities for capturing screen images. These can be used directly with **Ptolemy**. However, colors that work well on the screen are not always ideal for hardcopy. For this reason, two sets of alternative colors have been devised. The default works well with color printers. The second works well with black and white printers. For black and white printers, set the PIGIBW environment variable before running pigi, as in:

```
setenv PIGIBW
pigi
```

Now, the screen capture command can be used effectively. For example, under X11R4 or X11R5, the following command will print a window on a black and white postscript printer:

```
xwd | xpr -width 4 -portrait -device ps -gray 4 | lpr
```

Other alternatives include a program called xgrabsc. A simple use of this is to generate an encapsulated postscript image using the following command

```
xgrabsc -eps -page 4x2 -o mySchematic.ps
```

where mySchematic.ps is the name of the file into which you would like to store the EPS image. Then with the left mouse button, draw a box around the desired portion of the screen. This command will then save an encapsulated postscript file four inches by two inches called mySchematic.ps. This file can then be used an a wide variety of document processing systems. For example, to include this postcript in a TeX document, include the command

```
\include{psfig}
```

in the TeX file and use the commands

```
begin{figure}
  \centerline{
  \psfig{figure=mySchematic.ps,width=4in,height=2in}}
  \caption{Ptolemy Schematic}
\end{figure}
```

to display the postscript as a figure within a document.

The document processing system FrameMaker has built-in facilities to capture a region of the screen and import it into a document. FrameMaker can also import and print ordinary color X window dumps. To have these displayed in color on the screen, the following lines should appear in your X Resources file:

```
Maker.colorImages:      True
Maker.colorDocs:        True
```

The FrameMaker utility "fmcolor" can be used to reduce a color window dump to a black and white window dump. This will save space and avoid any dithered imitations of color. The color window dumps gotten with PIGICP set can be converted to black and white window dumps using the following FrameMaker command:

```
fmcolor -i 90 filein fileout
```

A useful hint when using such a document editor is to turn off the lables in pigi before capturing the image, and then to use the document editor itself to annotate the image. The fonts then will be printer fonts rather than screen fonts. To turn off the labels, execute the Vem command "layer-display" under the Vem "Options" menu. FrameMaker can also, in principle, import postscript files generated in the process of executing the ‘‘print facet’’ command. However, the results are not usually satisfactory, so this method is discouraged.

To restore normal color, simply exit pigi, and unset the environment variable:

```
unsetenv PIGIBW
```

Next time you run pigi, it will have whatever colors you originally specified in your X resources file, or if none, the default colors.

2.9. Other Useful Information

In this section we cover additional information which may be useful. More topics will be covered in following sections.

2.9.1. Plotting Signals and Fourier Transforms

The **Ptolemy** menu has a submenu called *Utilities* that invokes some useful, frequently used, predefined universes. For example, the *plot signal* (‘‘~’’) command will plot a signal. The signal can be read from a file, or specified using the syntax for specifying the value of a *floatarray* parameter in pigi. For example, if the value of the *signal* parameter is:

```
1 [10] -1 [10]
```

then the *plot signal* command will plot ten points with value 1.0 and ten points with value -1.0. The *options* parameter can accept any options understood by the xgraph program (see the xgraph section of the Almagest). To plot a signal stored in a file, simply use the following syntax for the *signal* parameter:

```
< filename
```

Another useful *Utilities* command is *DFT* (‘‘^’’), which reads a signal just as above and plots the magnitude and phase of the discrete-time Fourier transform of the signal. These are plotted as a function of frequency normalized to the sampling rate, from -(PI) to PI. The sampling frequency is assumed to be 2 PI. A simple phase unwrapping algorithm is used to give more meaningful phase plots. A radix-2 FFT is used, so the order of the Fourier transform must be a power of two. The order can be longer than the signal, in which case, zero-padding will occur.

2.9.2. Setting parameters

The *edit-params* or *define-params* commands in pigi permit the user to set the initial value of a settable state of any star, and to define parameters for a galaxy or universe. The syntax for linking the values of the star parameters to values of galaxy or universe parameters is simple. Furthermore, a fairly general expression language is supported.

Parameter values set through pigi can be expressions. This is particularly useful for propagating values down from a universe parameter to star parameters somewhere down in the hierarchy. An example of a valid parameter expression is:

```
PI/(2*order)
```

where **`order`** is a parameter defined in the galaxy or universe. The basic arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (ˆ). These operators work on integers and floats. Currently all intermediate expressions are converted to the type of the parameter being computed. Hence, it is necessary to be very careful when, for example, using floating point values to compute an integer parameter. All intermediate expressions will be converted to integers.

When defining complex values, the basic syntax is

```
(real, imag)
```

where **`real`** and **`imag`** evaluate to integers or floats.

When defining arrays of integers, floats, or complex numbers, the basic syntax is a simple list separated by spaces. For example,

```
1 2 3 4 5
```

defines an integer array with five elements. The elements can be expressions if they are surrounded by parentheses:

```
1 2 PI (2*PI)
```

Repetition can be indicated using the following syntax:

```
value [n]
```

where **`n`** evaluates to an integer. An array or portion of an array can be input from a file, using the symbol ‘‘<’’ as in the following example:

```
1 2 < filename 3 4
```

Here, the first two elements of the array will be 1 and 2, the next elements will be read from file **`filename,`** and the last two elements will be 3 and 4. This latter capability can be used in combination with the **`WaveForm`** star to read a signal from a file.

For string valued parameters, most of the above expressions will not work. Furthermore, there is a bit of complication when one wishes to set a string parameter equal to the value of a galaxy or universe parameter, because a distinction must be made between a sequence of characters that give the name of a symbol and a sequence of characters to be interpreted literally. The syntax to use is explained in the example:

```
This string has the word {word} taken from another parameter
```

Here, **`{word}`** represents the value of a string universe or galaxy parameter.

2.9.3. Particle Types

The packets of data that pass from one star to another in **Ptolemy** are called *particles*. So far, all particles have been simply floating point numbers representing samples of signals. However, several other data types are supported. Each star icon has a protrusion for each porthole. In pigi, if you are using a color monitor, the color of the stem indicates the type of data that the porthole consumes or produces. A blue stem on an input or output of a star icon indicates type "float", a white stem indicates type "complex", a green stem indicates "packet", and a red stem indicates "anytype".

Ptolemy usually make conversions between these types automatically. The float to complex conversion does the obvious thing, putting the float value into the real part of the complex, and setting the imaginary part to zero. The complex to float conversion computes the magnitude of the complex number. The XGraph star, and most other stars that generate output, accept "anytype" of input. If the input is complex, the magnitude will be plotted, because it will be cast to a float internally. If you want to plot both the real and imaginary parts, you should use the ComplexReal conversion star first.

For technical reasons too involved to explain here, automatic type conversions cannot always be made. In particular, Ptolemy can only do one type conversion for any given output of a star. So if a single output is connected to multiple inputs of different types, Ptolemy may complain. For example, suppose you connect a "float" output to both a "complex" input and a "float" input. Ptolemy will complain. A simple solution is to insert a unit gain (use the FloatGain star) between the "float" output and the "complex" input. Then the type conversion occurs on the output of the FloatGain star, which goes only to a "complex" input.

Automatic type conversions have one potentially serious pitfall. Suppose you connect a "float" output to a "complex" and "anytype" input. This is legal, because Ptolemy can satisfy the constraints with just one type conversion, float to complex. However, suppose the "anytype" input belongs to an XGraph star. As explained above, the XGgraph star plots the magnitude of a complex number, unless you have an explicit ComplexReal conversion star. Hence, instead of plotting the "float" output, XGraph will plot the magnitude, converting all negative numbers to positive. Inserting a ComplexReal is one way to correct the problem. Another way is to use the FloatGain star with unity gain to postpone the conversion to complex on the other fork.

2.9.4. Moving objects

Sometimes you may want to move objects around within your schematic. Use the VEM command *move-objects* (‘‘m’’) in the *Selection* menu to do this. You can move objects as follows:

- Select the objects that you want to move.

- Using the right mouse button, drag the objects to the desired location.
- Execute *move-objects*, ‘‘m’’.

2.9.5. Copying objects

You can create a new instance of any object in a facet by placing a point where you want the new instance, moving the mouse to the object you wish to copy, and executing ‘‘create’’ (‘‘c’’). However, this does not copy the parameter values. If you wish to create a new instance of a star or galaxy that has exactly the same parameter values as an existing instance, you should use the *copy-objects* (‘‘x’’) command in the VEM ‘‘Selection’’ menu. To do this, first select the object or objects you wish to copy. Then place a point in the center of the object. Then place a second point in the location where you would like the new object, and type ‘‘x’’. The new object starts life selected, so you can immediately move it, or type ‘‘control-u’’ to unselect it. As of this writing, VEM unfortunately does not allow you to copy objects from one facet to another.

2.9.6. Labeling a Design

It is useful to annotate a block diagram with titles and comments. The VEM *edit-label* (‘‘E’’) command in the ‘‘Edit’’ menu will do this. It takes two arguments: a point specifying the position of the label, and the name of a *layer*, which determines the color of the label. Place a point where you would like the label, and then type a layer name, such as "whiteSolid" (with the quotation marks). Then type ‘‘E’’. A dialog box will appear, offering various options. Type the text for your label in the ‘‘Label’’ box. It can contain carriage returns. To select a text height (font size) you can move the slider to the right of the ‘‘Text Height’’ box. The middle button moves the slider by large amounts, and the left and right buttons are used for fine tuning. The default, 100, is too big for all but the loudest titles. Sizes 60 and 40 work well with the overall scaling of **Ptolemy** facets. You can also change the justification by clicking the left button to the right of each justification box. A popup menu lists the options.

To determine which layers (colors) are available for text labels, open the layers palette. This can be done with the VEM *palette* (‘‘P’’) command in the ‘‘System’’ menu. A window will pop up showing the available colors. The names of the colors appear at the top of each column of boxes. The layers "redSolid" and "blackSolid" work well for labels. The layers "redOutline" and "blackOutline" are similar, except that a box will be drawn around the label.

2.9.7. Icon Orientation

Most **Ptolemy** icons have inputs coming in from the left and outputs going out to the right. To get better looking diagrams, you may sometimes wish to reorient the icons. This can be done with the VEM command ‘‘transform’’ (‘‘t’’). Select the icon you wish to transform and type ‘‘t’’ as many times as necessary to get the desired orientation. Each time, you get a 90 degree rotation. Then execute the move-object ‘‘m’’ to commit the change. Notice that a 180 degree rotation results in an upside down icon. To avoid this, reflect the icon rather than rotating it. To reflect it in the vertical direction (exchanging what’s on top for what’s on the bottom), select the object, type "my" (include the quotation marks), type ‘‘t’’ to transform, and ‘‘m’’ to move. To reflect along the horizontal direction, use "mx" instead of "my".

2.9.8. Finding the names of terminals

Some stars have several terminals, each with a different function. The documentation may refer to these terminals only by name. The name of a terminal is not normally visible when an icon is viewed with normal scaling. However, zooming in will eventually reveal the name. The easiest way to do this is to draw a box around the terminal and open a new window with the ‘‘o’’ command. Then you can zoom in here if necessary.

2.9.9. Multiple inputs and outputs

Ptolemy supports star definitions that do not specify how many inputs or outputs there are. The **`Add`** and **`Fork`** stars are defined this way, for instance. Take a close look at the third star in the third row of the SDF main.pal palette. It is an adder with only one input terminal. However, the input terminal is unusual in that it has two arrow heads. This signals a multiple input. Any number of outputs can be connected to this terminal (subject to restrictions explained below), and each of those outputs will be added.

Sometimes, such terminals are not convenient. It is better to have a set of icons for the commonly used numbers of terminals. The icon two to the left of the multi-input **`FloatAdd`** is also a **`FloatAdd,`** but it has an icon that shows only two normal inputs. The **`fork`** stars in the sixth row of the palette are similarly just different icons for the same star. It is easy to create such icons for any existing star.

Suppose you wish to create an icon for an adder with four inputs. In any facet, execute the pigi command ‘‘Extend:make-star’’ (‘‘*’’). A dialog box appears. Enter ‘‘FloatSum.input=4’’ for the star name, ‘‘SDF’’ for the domain, and ‘‘~ptolemy/src/domains/sdf/stars’’ for the star src directory (this is where the source code is stored). Note that ‘‘input’’ is the name of the particular multiple input that we want to specify. If you do not know where the source code is stored, then just look-inside (‘‘i’’) an existing instance of the star. The VEM console window and the header of the editing window that open both tell where the star source code is. A second dialog box appears asking you where you would like the icon put. Accept the default, ‘‘./user.pal’’. Then open user.pal using ‘‘O’’ to see the new four-input adder icon. You may edit this icon, as explained below.

2.9.10. Using delays

In the SDF and DDF domains, delays can be placed on arcs. A delay in these domains is not a star, but rather is a property of the arc connecting two stars. The interpretation of the delay is as an initial particle on the arc. To use these delays in pigi, the user places a delay icon on top of the wire connecting two instances. The delay icon is a green diamond in the SDF and system palettes. You can specify the number of delays by executing *edit-params* with the cursor on top of the delay icon.

2.9.11. Auto-forking

In pigi, a single output can be connected to any number of inputs, as one would expect. The interpretation in most domains is that the one output is broadcast to all the inputs. There are several point-to-point connections, therefore, represented by the net.

However, there are restrictions. To understand these restrictions, it is worth explaining that VEM stores connectivity information in the form of netlists, simply listing all terminals that are connected together. If a delay appears somewhere on a net, and that net has more than one point-to-point connection, then it is not easy to determine for which connection(s) the delay is intended. Consequently, at the time of this writing, delays are disallowed on nets with more than one connection. If you attempt to put a delay on such a net, then when you try to run the system, an error message will be issued, and the offending net will be highlighted. To get rid of the highlighting, execute the pigi command ‘‘Edit:clear-marks’’. To fix the problem, delete the offending net, and replace it with one or more *fork* stars and a set of point-to-point connections. All domains for which this problem occurs have fork stars.

There is also a more subtle restriction. Suppose two outputs are connected to a single multiple-input terminal. Then neither of these outputs can also be connected to some other input terminal. If it is, **Ptolemy** will issue the error message ‘‘multiple output ports found on the same node.’’ The reason this happens is simple. VEM knows nothing about multiple inputs, so it sees a net with more than one output and more than one input. **Ptolemy** is not given enough information to reconcile this and figure out which outputs should be connected to which inputs. To avoid this problem, it is again necessary to use explicit fork stars.

Both of the above restrictions may be eliminated in future versions.

2.9.12. Dealing with errors

Ptolemy is composed of several components of which pigi is one. When occur, it helps to know which component detected the error so that it can be corrected.

There are three main components involved in processing a schematic: VEM, pigi, and the Ptolemy kernel. Each component corresponds to one phase of processing:

VEM	Handles schematic entry and maintains net list information for the design. VEM also handles all accesses to Oct.
pigi	Translates schematic information that VEM produces into the textual language which the Ptolemy kernel expects.
Ptolemy kernel	Handles the execution of the design.

Errors can occur in any of these phases. When correcting errors, it helps to know in which phase the error occurs.

When errors occur in VEM, VEM prints the error in the console window. For example, if you enter a point argument and execute *create* when the cursor is not over an instance, then VEM displays the message ‘‘Cannot place contact’’. Usually, VEM errors are easy to fix. In this case, VEM expects the user to specify the instance to be created.

Pigi errors can occur when any of the pigi commands are invoked. When errors occur during compilation, pigi usually highlights in red the object in the schematic associated with the error. When this happens, you can execute the *clear-marks* command to clear the highlighting. If such an error occurs and the reason for the error is not obvious, try deleting the indicated objects and redrawing them.

2.9.13. Copying and moving designs

In one of our examples, we used ‘‘cp -r’’ to make a copy of a facet. In general, however, copying entire designs this way does not work. For it to work in the general case, you must also change some data in the facets that you copy. In particular, each facet has pointers to the icons it uses. If you move a galaxy, for example, then any pointer to the icon for that galaxy becomes invalid (or ‘‘inconsistent’’ in Oct terminology).

A utility program called ‘‘masters’’ has been provided for this purpose. This replaces the program from the Octtools distribution, called ‘‘octmvlib’’, that was used with earlier versions of **Ptolemy**.

Moving palettes, stars, galaxies, and universes

Palettes, star icons, galaxies, and universes are stored as Oct facets. Special care is required when moving or copying Oct facets. First, as emphasized before, every Oct facet is stored as a directory tree, so a copy should use "cp -r". Next, keep in mind that there may be pointers to the moved object in other facets. If you know where all these pointers might be, then moving facets is easy. If you do not know where all the pointers are, then your only practical choice is to leave a symbolic link in place of the old location pointing to the new.

Moving stars

Suppose you have developed a fantastic new star called alphaCentaur, and you wish to install it in a palette that is available for general use. Since you have developed the star, you know where it is used. The star icon itself is stored in two facets:

```
alphaCentaur/schematic/contents;
alphaCentaur/schematic/interface;
```

The first of these stores the schematic, the second stores the icon. The peculiar semicolon at the end is actually part of the file name. First move the source code:

```
mv alphaCentaur.pl destinationDirectory
```

Then edit the makefile or make.template in that directory to be able to compile the new star. Then move the icon:

```
mv alphaCentaur destinationDirectory
```

This moves the entire directory tree. You must now change all references to the icon so that they reflect the new location. Suppose you have a test universe called alphaTest. This should be modified by running the ‘‘masters’’ program as follows:

```
foucault.berkeley.edu 102: masters alphaTest
Running masters on wave
Pathname to replace (? for a listing):
```

Enter a question mark to get a list of all icons referenced in the facet:

```
Pathname to replace (? for a listing): ?

Pathnames currently found in the facet (minus connectors and delays):
  ~yourname/oldDirectory/alphaCentaur
        $PTOLEMY/src/domains/sdf/icons/Ramp
        $PTOLEMY/src/domains/sdf/icons/Sin
        $PTOLEMY/src/domains/sdf/icons/Xgraph
Pathname to replace (? for help):
```

The last three items are pointers to official **Ptolemy** icons. There is no need to change these. You should now enter the string you need to replace and the replacement value:

```
Pathname to replace (? for help): ~yourname/oldDirectory
New pathname: ~yourname/destinationDirectory
```

Next, use masters the same way to modify any palettes that reference the moved icon. For instance, the "user.pal" palette is a likely candidate. If you miss a reference, Oct will issue an error message when it tries to open the offending palette.

2.10. Defining New Stars

Only the most casual user will get by with the stars provided in the Ptolemy palettes. New stars are defined using the **Ptolemy** preprocessor language, which is described in its section in the manual. This pigi section will not attempt to teach you how to write your own stars.

There are two ways to use your own personal star with Ptolemy: use the dynamic linking facility, or build your own version of *pigiRpc*. If the incremental linking facility will suffice for your purposes, all you need to do is create an icon for the star; pigi will automatically link in the star when it is required. The incremental linking facility suffers from certain restrictions, however:

- The star may not call any functions except for those defined in the star's source file and those that are already present in pigi (you can count on all the kernel routines and all the domain-specific kernel support routines already being present in pigi).
- You may not redefine a statically linked star. (This restriction may be removed in the future).

If you wish to dynamically link a star is derived from another dynamically linked star, you must ‘‘permanently’’ link the parent star. This can be done with the ‘‘Extend:load-star-perm’’ command (‘‘K’’). To do this, place the mouse over an icon representing the parent star, and type ‘‘K’’. Once the parent star is permanently linked, it cannot be replaced or redefined: you must restart pigi.

You can make your own private version of *pigiRpc*; this is described in a later section.

It is a simple matter to create an icon for a new star; the procedure is the same for dynamically linked stars as it is for statically linked ones. The pigi command ‘‘Extend:make-star’’ (with accelerator ‘‘*’’) creates the icon. A dialog box will query you for the name of the star and the location of its source code. Please note that pigi requires the location be specified relative to your or some other user's home directory. For instance, if your login name is ‘‘joe’’, and the star is defined in ~joe/ptolemy/mystar.pl, then the src directory is ~joe/ptolemy. The master copy of the star icon to be generated by the *make-star* command is stored in a directory determined by pigi based on the following rule: if the star source directory ends in ‘‘.../stars’’ and there exists a ‘‘.../icons’’ directory, then put the icon in that ‘‘.../icons’’ directory. Otherwise, just put the icon in the same star source directory. It is your responsibility to make sure the star source code is indeed in the star source directory you enter into the dialog box. Otherwise, pigi will not be able to locate the star source code when executing the *look-inside* command. The way pigi does a *look-inside* on a star icon is based on a similar rule: if the directory where the master copy of the star icon is stored ends in ‘‘.../icons’’ then pigi looks for the star source code in the ‘‘.../stars’’ directory, otherwise it looks for the source code in the same directory (the same rule is used when locating source files for dynamic linking).

If the star you specify is not already loaded into pigi, it will be compiled and linked in. If the star source directory ends in ‘‘../src/stars’’, Pigi will build the object file in the directory ‘‘../obj.*arch*/stars’’, where *arch* is replaced by **`sun4`** or **`mips`**, depending on what your machine type is; this feature is designed for supporting networked clusters of dissimilar machines. Similarly, if the source directory ends in ‘‘.../src’’ the object file will be made in ‘‘.../obj.*arch*’’.

If there is a file named ‘‘Makefile’’ or ‘‘makefile’’ in the object file directory, pigi will run the *make* program, using the makefile to create the object file (or make sure it is up to date). If there is no makefile, pigi will run a make-like procedure on its own, running the preprocessor as needed to produce the C++ source files, then running the C++ compiler to create the object file. By default, the C++ compiler will be told to look for include files in the kernel directory and the domain-specific kernel and star directories; if this is not adequate, then you need to write a makefile. Once compilation (if any) is complete, the dynamic linker is used to load the star into the system. Compilation errors, if any, will appear in a popup window.

Pigi will next query for the name of a palette into which to store (an instance of) the icon. If the palette does not already exist, it will be created. The dialog box suggests using the palette ‘‘./user.pal’’, simply because this is easy to access via the *open-palette* (‘‘O’’) command. Note that you can have many palettes named ‘‘user.pal’’; the one opened by the *open-palette* command is the one in the directory where you started pigi.

After the icon has been created, you can use it immediately. The icon that pigi generates has a generic star symbol in it. You can change this symbol if you want; see the section below on editing icons.

An important note: whenever the definition of a star is changed so that the new definition has different I/O ports, the icon must be updated as well. You can do this by calling *make-star* again to replace the old icon with a new one.

If you modify the code for a dynamically loaded star and wish to load the new definition, execute the *load-star* command while pointing to an instance of the star (or type ‘‘L’’). This will not work if you previously permanently linked the star with the *load-star-perm* command (‘‘K’’ key). In this case, you must restart pigi to load in new version.

2.11. Editing Icons

Pigi automatically generates icons for stars and galaxies, respectively, when you invoke the *make-star* or *make-schem-icon* command from the Extend menu. Pigi puts the new icon in a user-specified palette, which by default is user.pal in the directory in which you started pigi. More or less any VEM manipulations can be performed on this icon, but some guidelines should be followed. These icons have a generic symbol. To change it, place the cursor over the icon and execute the *edit-icon* (‘‘I’’) command in the pigi menu.[4] A new window containing the icon facet will appear.

Remember that icons are stored in interface facets and that the icons which in contents facets are really instances of icons. These instances merely refer to the actual icon facet. The *edit-icon* command opens a window into the actual icon. Any changes made in this window will affect the appearance of all instances referring to the icon.

Recall also that icon facets store a different kind of data from other facets. Icon facets contain information that tells VEM how to draw objects. Hence, a different set of commands must be used to edit icons. Whenever you edit an icon, VEM switches to a different mode called ‘‘physical editing style’’. In this mode, we create objects such as lines, boxes, and polygons. This is in contrast to ‘‘schematic editing style’’ which we used before to create instances and connect them together with wires. Physical editing style shares many commands with schematic editing style. For example, *select-objects* is active in both modes. A list of useful physical editing style commands and their key bindings are given in appendix B.

The commands that create geometry expect a layer argument. The layer an object is attached to determines its color and its fill. To specify a layer, place the cursor over an object attached to the desired layer before executing a command. You can open a palette of layers with the *palette* (‘‘P’’) command.

The layer palette contains several columns of solid and outline colors, with the name of the color at the top of the column. Colors at the top of each icon will be layered on top of colors below them in the columns. A set of special layers are arranged at the bottom of the palette. The layers for icon stems are explained below. The layers for icon bodies define the icon background and optional icon shadow.

A few simple notes will greatly help. First note that when the icon window is opened, the snap is automatically set to 5 units. This is because the default snap for schematic windows, normally 25 units, is far too coarse for most icon editing functions. A reasonable compromise is 5 units, unless you are going to try to create a very elaborate icon, in which case 1 unit is probably what you want. The VEM Options:window-options command allows you to change the snap.

When editing an icon, the VEM menu is different than when you are editing a schematic. This is because you are working with the physical view of a facet. You can explore most of the VEM commands in the Edit and Selection VEM menus. Most icons can be created by experimenting with the following operations:

1) Select the default symbol (a star, galaxy, or cluster of galaxies), by drawing a box (drag the left mouse button) and typing ‘‘s’’ (or using Selection:select-objects in the menu). You can

---

[4] You must have write permission on the facet to change the icon.

unselect with ‘‘u’’ or control-u. An alternative selection method is to place a point and type ‘‘s’’. This usually provokes a dialog box to resolve ambiguities. Delete whatever parts of the icon you don’t want using ‘‘D’’ or Selection:delete-objects. WARNING: Do not delete terminals! If you accidentally delete a terminal, the easiest action is begin again from scratch, asking pigi to create a new icon.

2) Bring up the VEM color palette by typing ‘‘P’’ (or System:palette in the menu).
3) Draw a line by clicking the left button to place a point, and pushing and dragging the mouse button from the same point. Then move the mouse to desired color in the palette and type ‘‘c’’ (or Edit:create-geometry from the menu). A line may consist of multiple line segments by just repeatedly pushing and dragging the mouse button.
4) To create filled polygons, place points at the vertices, then type ‘‘c’’ on the appropriate solid color in the palette.
5) To create a circle, place a point at the center, a point on periphery, and type ‘‘C’’ on the appropriate color. To create a filled circle, use a line segment instead of a pair of points.
6) Objects can be moved by selecting them, dragging the mouse (using the right button) to produce an image of the object in the desired place, and typing ‘‘m’’.
7) You may change (or delete) the label that pigi automatically puts at the bottom of the icon. To change it, select it and type ‘‘E’’ (Edit:edit-label in the menu). The resulting dialog box is self explanatory. The standard Emacs-like editing commands apply.
8) BE SURE TO SAVE YOUR ICON. This can be done by typing ‘‘S’’ (System:save-window in the VEM menu). You can close your window with control-d. Note that VEM buffers the data in the window. You can close it and reopen it without saving it, as long as the session has not been interrupted.

By convention, the data types supported by a terminal are indicated by the color of the stem that connects the terminal to the body of the icon. The following colors are currently in use:

ANYTYPE: red
FLOAT: blue
INT: yellow
COMPLEX: white
PACKET: green

The color is currently set automatically by the icon generator by using layers defined specifically for this purpose, called anytypeColor, floatColor, intColor, complexColor, and packetColor. You can change the color manually, if you wish. To do this, select the stem, type ‘‘"xxxx"’’, where xxxx is replaced by the color name (the quotes are necessary), and then type the single character ‘‘l’’ (an el — or Edit:change-layer in the menu). Again, be sure to save the window. To see what color names are available in the current technology, execute the VEM ‘‘System:palette’’ command, or ‘‘P’’. Note the layers specifically used for icon stems.

One final editing operation is a little trickier: moving terminals. Pigi places terminals rather arbitrarily, since it knows nothing of their function. You may wish to have a smaller icon than the default, in which case you have to move the input terminals closer to the output terminals. Or may wish to change the order of the terminals, or you may to have terminals on the top or bottom of the icon rather than right or left. All of these can be done, but the following cautions must be observed:

1) Do not move terminals of icons that have already been used in applications. Unfortunately, if you do this, the block will become disconnected in all applications where it is used. If you are tempted to move terminals in a commonly used block, consider the tedium of finding all applications (belonging to all users) and reconnecting the block. ONLY MOVE TERMINALS ON BRAND NEW ICONS.
2) You must respect the default snap of 25 for schematic windows, and move terminals to a point that falls on a multiple of 25 units. Otherwise, connecting to the terminal will be very difficult. Normally, when the icon window opens, the grid lines are 10 units apart. So you can place terminals any multiple of 2.5 grid lines away from the center.
3) To orient terminals so they are aiming up or down, select the terminal, type ‘‘t’’ (or Selection:transform in the menu), then type ‘‘m’’ to move it. Repeatedly typing ‘‘t’’ will continue to transform the terminal.

A little more detail on the Oct internals may be useful if you explore the files that are created by these operations. For make-schem-icon, if the schematic is called *xxx*, then the icon itself is stored in ’’*xxx*/schematic/interface;’’. The semicolon is part of the filename (this creates some interesting challenges when manipulating this file in Unix, since Unix interprets the semicolon as a command delimiter). The standard stars that are normally part of the Ptolemy distribution are stored in ‘‘˜ptolemy/src/domains/*dom*’’, where *dom* is the domain name such as ‘‘sdf’’ or ‘‘de’’. The icons for the stars are stored in a subdirectory called **`icons`**, the icons for demo systems in a subdirectory called **`demo`**, and the source code for the stars are stored in a subdirectory called **`stars`**. Feel free to explore these directories.

2.12. Multi-Domain Universes

The domain of an application is set using the pigi ‘‘Edit:edit-domain’’ or ‘‘d’’ command. This command causes a checklist to appear listing all domains currently linked into the system. All examples in the SDF Demo palette are one-domain applications, using only SDF. Several examples of multi-domain applications can be found in the DDF and DE Demo palettes. It is instructive to explore these applications, using the edit-domain command at all levels of the hierarchy to see what domains are used.

Recall that a **`Wormhole`** in **Ptolemy** is a block that has a different domain on the outside than on the inside. In pigi, wormholes look exactly like galaxies. The only difference is that the domain is different on the inside than on the outside.

To build multi-domain applications, it is necessary to understand the models of computation in each domain, to ensure that application will behave consistently at the domain boundaries. For this, it is necessary to refer to the domain documents, included in the Almagest.

2.13. 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 the section on writing stars in the Almagest). 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 don’t have to wait for the dynamic linking to complete every time you execute your applications.

The **Ptolemy** kernel and VEM are separate Unix^5 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.

The instructions for creating your own version of pigiRpc are contained in the file

```
~ptolemy/src/pigiRpc/MakeYourOwn
```

These instructions basically consist of copying a small set of files into a directory where you want your private version of pigiRpc to reside, say ~/PTOLEMY/MYPIGI, and editing these files. By editing the makefile, you can customize the configuration, specifying your own code that you want linked in, and removing domains that you do not want linked in. To remove a domain, simply avoid linking any stars from that domain. You can easily create, for example, a version of pigiRpc that contains only the SDF domain. After editing these files according to the instructions, you must make pigiRpc:

```
cd ~/PTOLEMY/MYPIGI
make pigiRpc
```

Alternatively, you can make a version of pigiRpc with debug symbols:

```
make pigiRpc.debug
```

Then you should set the environment variable PIGIRPC to point to your private version of pigiRpc. If you use csh:

```
setenv PIGIRPC ~/PTOLEMY/MYPIGI/pigiRpc
```

or

```
setenv PIGIRPC ~/PTOLEMY/MYPIGI/pigiRpc.debug
```

The next time you run pigi, it will use your private version of pigiRpc. To revert to using the installed pigiRpc, just type

```
unsetenv PIGIRPC
```

2.14. 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^6 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:

Unix is a trademark of AT&T.

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

The message

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

will 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’’. This file is useful for finding the problem.

2.14.1. 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 symbolic debugger (gdb) will show the state of the stack at the point where the program failed. The most recently called function might give you a clue about the cause of the problem. Here is a typical session:

```
foucault.berkeley.edu 54: gdb pigiRpc
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 4.7, Copyright 1992 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) core core
Core was generated by ‘pigiRpc’.
Program terminated with signal 11, Segmentation fault.
#0  0x2eebb4 in end ()
(gdb) where
#0  0x2eebb4 in end ()
#1  0x1c774c in SimControl::doPreActions ()
#2  0x1c8538 in Star::run ()
#3  0x16ee74 in DataFlowStar::run ()
#4  0x16d888 in SDFScheduler::runOnce ()
#5  0x16d7c4 in SDFScheduler::run ()
#6  0x1cb4c0 in Target::run ()
#7  0x1cd4e4 in Runnable::run ()
#8  0x1bd810 in InterpUniverse::run ()
#9  0x197774 in KcRun ()
#10 0x18edac in RpcRun ()
#11 0x18eeb8 in RpcReRun ()
#12 0x177634 in RPCApplicationProcessEvents ()
#13 0x17675c in RPCMain ()
#14 0x2300 in main ()
(gdb)
```

The ‘‘where’’ command shows that state of the stack at the time of the crash. 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.

2.14.2. More Extensive Debbugging

To do more extensive debugging, you need to create or find a version of pigiRpc with debug symbols, called pigiRpc.debug. See the previous section for instructions on how to do this. Once you have this, and have set the PIGIRPC environment variable to point to it, run pigi as follows:

```
pigi -debug
```

An extra window running gdb appears. (If this fails, then gdb is probably not installed at your 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 4.7, Copyright 1992 Free Software Foundation, Inc...
Breakpoint 1 at 0x22b0: file pigiMain.cc, line 45.

Breakpoint 1, main (argc=10, argv=0xf7fff77c) at pigiMain.cc:45
45              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 11, Segmentation fault
0x166d8c in end ()
(gdb)
```

At this point you can again examine the stack. This time, however, there will be more information. For same example as above:

```
(gdb) where
#0  0x166d8c in end ()
#1  0xa34ec in SimControl::doPreActions (which=0x2671c0)
    at ../../src/kernel/SimControl.h:89
#2  0xa42d8 in Star::run (this=0x2671c0) at ../../src/kernel/Star.cc:68
#3  0x5632c in DataFlowStar::run (this=0x2671c0)
    at ../../../../src/domains/sdf/kernel/SDFStar.cc:81
#4  0x54d40 in SDFScheduler::runOnce (this=0x2681d8)
    at ../../../../src/domains/sdf/kernel/SDFScheduler.cc:119
#5  0x54c7c in SDFScheduler::run (this=0x2681d8)
    at ../../../../src/domains/sdf/kernel/SDFScheduler.cc:98
#6  0xa7260 in Target::run (this=0x266f00) at ../../src/kernel/Target.cc:125
#7  0xa8eb4 in Runnable::run (this=0x266c50) at ../../src/kernel/Universe.h:
#8  0x9c1f8 in InterpUniverse::run (this=0x266bf8)
    at ../../src/kernel/InterpUniverse.h:64
#9  0x7eb2c in KcRun (n=10) at ../../src/pigilib/kernelCalls.cc:458
#10 0x75e80 in Run (facetPtr=0xf7fff4c8) at ../../src/pigilib/exec.c:164
#11 0x761c0 in RpcRun (spot=0xf7fff5e4, cmdList=0x164158, userOptionWord=0)
    at ../../src/pigilib/exec.c:258
#12 0x5eaec in RPCApplicationProcessEvents ()
#13 0x5dc14 in RPCMain ()
#14 0x2300 in main (argc=10, argv=0xf7fff77c) at pigiMain.cc:50
(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  0xa34ec in SimControl::doPreActions (which=0x2671c0)
    at ../../src/kernel/SimControl.h:89
89                              : !haltRequested();
(gdb) up
#2  0xa42d8 in Star::run (this=0x2671c0) at ../../src/kernel/Star.cc:68
68              go();
```

At this point, you can see that line 68 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) print *this
$1 = {<Block> = {<NamedObj> = {nm = 0x267268 "BadStar1", prnt = 0x266bf8,
      myDescriptor = 0x166ca0 "Causes a core dump deliberately.",
      _vptr$FOO = 0x166e08}, ports = {<NamedObjList> = {<SequentialList> = {
          lastNode = 0x267258, dimen = 1}, }, },
    states = {<NamedObjList> = {<SequentialList> = {lastNode = 0x0,
          dimen = 0}, }, },
    multiports = {<NamedObjList> = {<SequentialList> = {lastNode = 0x0,
          dimen = 0}, }, }}, targetPtr = 0x266f00, 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.

2.15. Where to Find More Information

To find out more about Ptolemy and its capabilities, see the Almagest, or complete Ptolemy User's Manual [1]. This document is a part of that manual, but may also be distributed or printed separately. Pigi uses both VEM and Oct. Reasonably complete VEM information is included in the VEM section of the Almagest. Further information can be found in the VEM manual [3]. Information about Oct is available in the Oct user's guide [4].

2.16. References

1. *Almagest: Ptolemy User's Manual*, University of California, Berkeley: Electronic Research Laboratory.
2. D. S. Harrison, *VEM Version 6-0 Manual*, University of California, Berkeley: Electronic Research Laboratory.
3. R. L. Spickelmier, *A User's Guide to Oct 2.0*, University of California, Berkeley: Electronic Research Laboratory, July 1989.
4. *Oct Tools Distribution 3.3 Manual*, University of California, Berkeley: Electronic Research Laboratory, 1989.

Appendix A: Pigi Command Summary

This appendix contains a list of all the **Ptolemy** commands and a brief description of what they do. The next appendix summaries VEM commands. To get the **Ptolemy** menu, hold the shift button, and click the middle mouse button. The menu is sticky, in that it stays around after you release the mouse button. The first menu you see lists only command categories. The submenu is gotten by moving the mouse to the arrow. The submenus have the executable commands. To the left of each command in the menu is the single key accelerator used for that command. These accelerators are case sensitive.

To execute a command, click any button again with the command highlighted. To remove the menu without executing the command, click any button with the mouse cursor somewhere away from the menu. The commands below are grouped according to the submenus in which they appear.

Ptolemy commands

Command	Key	Description
Edit Menu:		
edit-params	e	To edit the actual parameters of a star, galaxy, or delay, place the cursor over the icon of the desired object. To edit the formal parameters of a galaxy or universe, place the cursor in a window that contains the galaxy schematic and make sure the cursor is not over any instance.
define-params	+	This command allows you to change the list of formal parameters of a galaxy or universe, as well as the type and value. You can add or remove parameters from the list.
edit-domain	d	To specify the domain of a galaxy. If the domain is different from the domain where the galaxy is used, a wormhole will be automatically created. All domains currently linked to pigi will be offered as options.
edit-target	T	To specify a target architecture for a code generation domain. If no target architectures are defined for the current domain, an error message will be issued.
edit-comment	;	Append a comment to a galaxy or universe. For a galaxy, the comment will form the descriptor field for the Other:profile command.
edit-seed	#	To specify the seed of the random number generation.
find-name		Some error messages give the name of the offending instance of a block. This command will cause that block to be highlighted, given its name.
clear-marks		Pigi tries to highlight objects when an error occurs during compilation or execution, or when the find-name command is invoked. This clears the highlighting.
Window Menu		
open-palette	O	Open a palette from a list of palettes.
open-facet	F	Open an Oct facet. A new window is created displaying the facet. Pigi commands can be executed from this window.
edit-icon	I	Opens the facet containing the icon data for the instance under the cursor.
look-inside	i	Opens the facet containing the schematic data or the star definition for the instance under the cursor.

Ptolemy commands (continued)

Command	Key	Description
Exec Menu:		
run	**R**	Runs a universe. Cursor must be inside a universe facet.
re-run	**!**	Runs a universe again with the same parameters given on the last *run*.
compile-facet		Compiles the schematic which is under the cursor. The compile is recursive and so if the schematic contains sub-galaxies, each sub-galaxy schematic gets compiled also. Average users will not need this command.
display-schedule		Displays a parallel schedule if supported by the current Target.
Extend Menu:		
make-schem-icon	**@**	Make an icon for the schematic under the cursor.
make-star	*****	Make an icon for a new star.
load-star	**L**	Load the star pointed to by the cursor.
load-star-perm	**K**	Permanently load a star so that you can subsequently load stars derived from it.
Filter Menu:		
equirriple FIR	<	This is a provisional filter design capability, which simply invokes a public domain Fortran program in its own window for designing FIR filters using the Parks-McClellan algorithm. More complete documentation is in the filter section of the Almagest. Note that this program starts in your home directory, so filenames should be given relative to the home directory.
window FIR	>	Like the equirriple FIR, this is a provisional filter design capability, which simply invokes a public domain Fortran program in its own window. It designs FIR filters using the windowing technique. More complete documentation is in the filter section of the Almagest. Note that this program starts in your home directory, so filenames should be given relative to the home directory.
Utilities Menu:		
plot signal	**˜**	Plot a signal.
plot Cx signal	**-**	Plot real and imaginary parts of a complex signal.
DFT	**ˆ**	Plot the discrete-time Fourier transform (magnitude and phase) of a signal, as a function of normalized frequency.
DFT of a Cx signal		Plot the discrete-time Fourier transform (magnitude and phase) of a complex signal, as a function of normalized frequency.
Other Menu:		
man	**M**	Display a section of the Almagest. Not functional at this time.
profile	**,**	Execute the Ptolemy profile command on the object under the cursor. The information currently appears in the VEM command window.
options		Edit various Pigi options.
version		Shows which pigiRpc executable is currently running.
exit-pigi		Clean exit from pigi application program. VEM continues to run.

Appendix B: VEM Command Summary

This appendix contains a summary of VEM commands and dialog box editing control characters. The VEM menu is obtained by clicking the middle mouse button in any facet window. Its structure is similar to that of the **Ptolemy** command menu. The commands are organized according to the submenu used to access them. The submenu names are listed first in each section. More complete documentation can be found in the VEM section of the Almagest.

VEM commands

Command	Key	Description
No menu entry:		
	ctrl-H, del	Clear last argument on the argument stack.
	ctrl-U	Clear all arguments on the argument stack.
redraw-window	ctrl-L	Redraws the window.
System menu:		
open-window	o	Opens a new window.
close-window	ctrl-D	Closes a window (doesn’t save).
where	?	Shows where your cursor is
palette	P	Open the layer palette.
save-window	S	Saves contents of window.
bindings	b	Identify key bindings. After issuing the bindings command, type the key you are interested in. VEM will identify the command associated with it.
re-read		Re-reads a facet from disk.
Display menu:		
pan	p	Move area under cursor to center of window.
zoom-in	z	Zooms in one level.
zoom-out	Z	Zooms out one level.
show-all	f	Zoom to full view of facet.
same-scale		Force a window to have the same scale as some other window. Place a point in the window with the right scale, and issue the same-scale command in the window you wish to rescale.
Options menu:		
window-options		Set various VEM options, including the snap.
layer-display		Select layers to display or omit.
toggle-grid	g	Turn on or off the grid display
Props&Bags menu:		
show-property	ctrl-P	Shows properties of object under cursor.
Undo menu:		
undo	U	Undo last action. Has infinite memory.
Edit menu:		
create	c	Create instances, wires.
delete-objects	D	Deletes selected objects.
edit-label	E	Edit a label.

VEM commands (continued)

Command	Key	Description
Selection menu:		
select-objects	s	Selects objects for manipulation.
select-net	ctrl-N	Select only nets.
unselect-objects	u	Unselects objects.
transform	t	Rotates an object 90 degrees. Use the move-instance or drag-instance command to finalize the transform. Use the "mx" or "my" modifier to reflect rather than rotate the object.
move-objects	m	Move selected objects, preserving their property lists.
copy-objects	x	Copy selected objects, preserving their property lists.
delete-objects	D	(repeated) Deletes selected objects.
Application menu:		
rpc-any		Start some other VEM application.

When editing a **Ptolemy** icon, the VEM menu is slightly different. The System, Display, Options, Props&Bags, Undo, and Application submenus are identical to those explained above. However, the Edit and Selection submenus are different. These are summarized below

VEM Physical Editing Commands (Edit and Selection)

Command	Key	Description
Edit menu:		
create-geometry	c	Create boxes, polygons, and paths (Oct terminology for a connected set of line segments). What gets created depends on argument types.
alter-geometry	a	Replaces a box, polygon, or path with a new one described on the argument list. E.g., to change the shape of a box, draw the new box, and execute this command.
change-layer	l	Place objects on a different layer (change color).
set-path-width	w	Change the width of lines drawn in particular layer.
create-circle	C	Create circles or arcs.
edit-label	E	Create or edit a label.
delete-physical	D	Remove specified object.
copy-objects	x	Copy objects.
Selection menu:		
select-objects	s	Selects objects for manipulation.
unselect-objects	u	Unselects objects.
move-physical	m	Move selected objects.
copy-physical	x	Copy selected objects.
transform	t	Rotates an object 90 degrees. Use the move-instance or drag-instance command to finalize the transform. Use the "mx" or "my" modifier to reflect rather than rotate the object.
select-terms	ˆT	select terminals.
delete-physical	D	Deletes selected objects.

For convenience, the Emacs-like commands used in all dialog boxes are repeated here:

Dialog Box Editing Control Characters

Key	Description
Delete, control-h	Delete previous character.
control-A	Move to beginning of line.
control-B	Move backward one character.
control-D	Delete next character.
control-E	Move to end of line.
control-F	Move forward one character.
control-K	Kill (delete) to end of line.
meta-B	Move backward one word.
meta-F	Move forward one word.
meta-H, meta-delete	Delete previous word.

3. ptcl: The Ptolemy Interpreter

Authors:

Joseph Buck

The **Ptolemy** interpreter, *ptcl*, provides a textual interface for constructing systems and performing simulations. It is based on John Ousterhout's Tcl, which is an extensible interpreter language. All the commands of Tcl are available in *ptcl*. This interface is more convenient than the graphical interface when large complex universes are being created automatically by some other program. It is also more convenient when using a symbolic debugger to debug a new piece of code linked to **Ptolemy**.

3.1. Getting Started

To get started, just make sure *˜ptolemy/bin* and *˜ptolemy/bin.ARCH* are in your PATH (where *ARCH* is replaced by ‘‘sun3’’ if you are on a Sun-3, ‘‘sun4’’ if you are on a Sun-4, ‘‘vax’’ if you are on a Vax, or ‘‘mips’’ if you are on a DecStation based on the MIPS architecture). Now type **`ptcl`** to invoke the Ptolemy interpreter. It is also possible to specify a file of interpreter commands as a command line argument; see the section ‘‘Loading Commands from a File’’ for details.

3.2. Using the Interpreter

There are two ways to use **Ptolemy**: an *interpreter* and a *graphical user interface*. The interpreter conveniently operates on dumb terminals and other environments where graphical user interfaces may not be available, and is described in this section. The graphical user interface is described in a separate section.

The **Ptolemy** interpreter, *ptcl*, accepts input from the keyboard, or from a file, or some combination thereof. It allows the user to set up a new simulation by creating instances of **`Star`**s, connecting them together, setting the initial value of internal parameters and states, running the simulation, restarting it, etc.

ptcl extends the Tcl interpreter language by adding new commands. The underlying grammar and control structure of Tcl are not altered. Commands in Tcl have a simple syntax: a verb followed by arguments. This document will not explain Tcl; please refer to the file ‘‘˜ptolemy/tcl/Tcl.man’’ which is included with this distribution; it is a copy of the Tcl Unix-style manual entry. This document describes only the extensions to Tcl made by *ptcl*.

The interpreter has a *known list* containing all the classes of **`Star`**s and **`Galaxy`**s it currently knows about. It is possible to add a new **`Star`** to the known list at runtime only by using the incremental linking facility, but this has restrictions, see the **`link`** command below (you can also make your own copy of the interpreter with your own **`Star`**s linked in). However, it's straightforward to add new classes of **`Galaxy`**s to the known list.

The interpreter also has a *current galaxy*. Normally, this is set to be the **`mainGalaxy`**. During the execution of a **`defgalaxy`** command, however, the *current galaxy* is set to be the galaxy being defined. After the closing curly brace of the **`defgalaxy`** command, the *current* *galaxy* is reset to be the `mainGalaxy`.

3.2.1. Commands for Defining the Simulation

This section describes commands that build the simulation and add `Star`s, `Galaxy`s, `State`s, and the connections among them.

Setting the Domain

Ptolemy supports multiple simulation domains, and so does the **Ptolemy** interpreter. Before creating a simulation environment and running it, it is necessary to establish the domain. Currently, nine domains are linked into the standard interpreter: synchronous data flow (SDF), discrete event (DE), and dynamic data flow (DDF), the Thor digital logic simulator (Thor), abstract code generation domains (CG and CG-DDF), and code generation domains for Motorola 56000 processors (CG56), Motorola 96000 processors (CG96), and C code generation (CGC). The interpreter has a *current domain*; this is initially `SDF`. The command

`domain` *domain-name*

changes the current domain; it is only legal when the current galaxy is empty. The argument must be the name of a known domain. The command

```
domain
```

returns the current domain. It is possible to create `Wormhole`s – interfaces between domains – by including a `domain` command inside a galaxy definition.

Creating Instances of Stars and Galaxies

The first step in any simulation is to define the blocks (`Star`s and `Galaxy`s) to be used in the simulation. The command

`star` *name class*

creates a new instance of a `Star` or `Galaxy` of class *class*, names it *name*, and inserts it into the current galaxy. Any `State`s in the `Star` (or `Galaxy`) are created with their default values. While it is not enforced, the normal naming convention is that *name* begin with a lower case letter and *class* begin with an upper case letter (this makes it easy to distinguish instances of a `class` from the `class` itself). The synonym

`galaxy` *name class*

is also provided to create an instance of a `Galaxy`.

Connecting Stars and Galaxies

The next step is to connect the blocks so that they can pass data among themselves using the `connect` command. This forms a connection between two `Star`s (or `Galaxy`s) by connecting their portholes. A porthole is specified by giving the `Star` (or `Galaxy`) name followed by the port name within the `Star`. The first porthole must be an output porthole and the second must be an input porthole. Example:

```
connect mystar output yourstar input
```

The connect command accepts an optional delay parameter:

```
connect mystar output yourstar input 1
```

specifies one delay on the connection. This parameter makes sense only for the dataflow and code generation domains; it is meaningless for DE. The delay argument may be an integer expression with variables referring to galaxy states as well.

One or both of the portholes may really be a `MultiPortHole`. If so, the effect of doing the connect is to create a new porthole within the `MultiPortHole` and connect to that (see also the `numports` command).

Netlist-style connections

As an alternative to issuing connect commands (which specify point-to-point connections) you may specify connections netlist-style. This syntax is used to connect an input to more than one output, for example (this is called *auto-forking*). Two commands are provided for this purpose. The `node` command creates a node:

node *nodename*

The `nodeconnect` command connects a porthole to a node:

nodeconnect *starname portname nodename*

Bus Connections Between MultiPortHoles

A pair of multiportholes can be connected with a bus connection, which means that each multiporthole has *N* portholes and they all connect in parallel to the corresponding port in the other multiporthole. The syntax for creating such connections is

busconnect *srcstar srcport dststar dstport width delay*

Here *width* is an expression specifying the width of the bus (how many portholes in the multiportholes); and *delay* is an optional expression giving the delay on each connection.

Connecting Internal Galaxy Stars and Galaxies to the Outside

When you define a new `Galaxy` there are typically external connections to that `Galaxy` that need to be connected through to internal blocks. The `alias` command is used to add a porthole to the current galaxy, and associate it with an input or output porthole of one of the contained `Star`s within the `Galaxy`. An example is:

```
alias galaxyin mystar starin
```

This also works if `starin` is a `MultiPortHole` (the galaxy will then appear to have a multiporthole as well).

Defining States for a Galaxy

A **`State`** is a piece of data that is assigned to a **`Galaxy`** and can be used to affect its behavior. Typically the value of a **`State`** is coupled to the **`State`** of blocks within the **`Galaxy`**, allowing you to customize the behavior of blocks within the **`Galaxy`**.

The **`newstate`** command adds a **`State`** of the given **`class`** to the current **`Galaxy`**. The form of the command is

`newstate` *state-name state-class default-value*

state-name is the name to be given to the **`State`**. *state-class* is the type of **`State`**; at present, the supported types are **`int`**, **`float`**, **`complex`**, **`intarray`**, **`floatarray`**, **`complexarray`**, and **`string`**. *default-value* is the default value to be given to the **`State`** if the user of the galaxy does not change it (using the **`setstate`** command described shortly). The *default-value* specifies the initial value of the state, and can be an arbitary expression involving constant values and other **`State`** names; this expression is evaluated when the simulation starts. The following state names are predefined: YES, NO, TRUE, FALSE, PI. YES and TRUE have value 1; NO and FALSE have value 0; PI has the value 3.14159... . Example: Some examples are:

```
newstate count int 3
newstate level float 1.0
newstate title string "This is a title"
newstate myfreq float galaxyfreq
newstate angularFreq "2*PI*freq"
```

The full syntax of state initial value strings depends on the type of state, and is explained in the manual section for the graphical interface (pigi).

Setting the Value of States

The **`setstate`** command is used to change the value of a **`State.`** It can be used in three contexts:

- Change the value of a **`State`** for a **`Star`** within the current galaxy.
- Change the value of a **`State`** for a **`Galaxy`** within the current galaxy.
- Change the value of a **`State`** within the current galaxy.

The latter would normally be used when you want to perform multiple simulations using different parameters.

The syntax of **`setstate`** is

`setstate` *block-name state-name value*

Here

- *block-name* is either the name of a **`Star`** or a **`Galaxy`** that is inside the current galaxy, and it is the block for which the value of the **`State`** is to be changed. It can also be **`this`**, which says to change a **`State`** belonging to the current galaxy itself.

- *state-name* is the name of a `State` which you wish to change.
- *value* is the new value for the `State`. The syntax for *value* is the same as described in the **`newstate`** command. However, the expression for *value* may refer to the name of one or more `State`s in the current galaxy or an ancestor of the current galaxy.

An example of the use of `setstate` is given in the section describing **`defgalaxy`** below.

Initializing all states

The **`initstate`** command (which takes no arguments) causes all states in the universe to be initialized (all the initial-value strings are parsed and interpreted). Note that this action is automatically taken whenever a universe is run or a schedule is generated; the only reason this command is provided is so that the user can check on the effect of provided state values with the **`show`** or **`dump`** commands.

Setting the Number of Ports to a Star

Some `Star`s in **Ptolemy** are defined with multiple-ports. These multiple-ports have an arbitrary number of connections, the number of connections being defined by the user of the `Star` rather than the `Star` itself. The **`numports`** command applies to `Star`s that contain `MultiPortHole`s; it causes a number of `PortHole`s to be created within the `MultiPortHole`. The syntax is

`numports` *star portname n*

where *star* is the name of a `Star` within the current galaxy, *portname* is the name of a `MultiPortHole` in the `Star`, and *n* is an integer, representing the number of `PortHole`s to be created. After the `PortHole`s are created, they may be referred to by appending *#n*, where *n* is an integer, to the multiporthole name, and enclosing the resulting name in quotes. The main reason to use this command is to allow the portholes to be connected in random order. Here is an example:

```
star summer Add
numports summer input 2
input galInput summer "input#1"
connect foo output summer "input#2"
```

Defining New Galaxies

The **`defgalaxy`** command allows the user to define a new class of `Galaxy`. The syntax is

`defgalaxy` *class-name* **{**
    command
    command
    ...
}

Here *class-name* is the name of the `Galaxy` type you are creating. While it is not required, we suggest that you have the name begin with a capital letter in accordance with our standard naming convention – class names begin with capital letters. *command* may be any of the commands described above – **`star`**, **`galaxy`**, **`connect`**, **`busconnect`**, **`node`**, **`nodeconnect`**, **`input`**, **`output`**, **`numports`**, **`newstate`**, or **`setstate.`** The defined class is added to the known list, and you can then create instances of it and add them to other galaxies. An example is:

```
defgalaxy SinGen {
     # The frequency of the sine wave is a state
     newstate freq float "0.05"

     star ramp Ramp
     # The ramp advances by 2*pi each sample
     setstate ramp step "6.283185307179586"

     # Multiply the ramp by a value, setting the frequency
     star gain Gain
     # The multiplier is set to "freq"
     setstate gain gain "freq"

     # Finally the sine generator
     star sin Sin

     connect ramp output gain input
     connect gain output sin input
     alias output sin output
}
```

In this example, note the use of **`State`**s to allow the frequency of the sine wave generator to be changed. For example, we could now run the sine generator, changing its frequency to "0.02", with the interpreter input:

```
galaxy generator SinGen
setstate generator freq "0.02"
star printer Printer
connect generator output printer input
run 100
```

You may include a **`domain`** command within a **`defgalaxy`** command. This creates an object known as a **`Wormhole`**, which is an interface between two domains. An example of this appears in a later section.

3.2.2. Showing the Current Status

The following commands display information about the current state of the interpreter.

Displaying the Known Classes

The `knownlist` command returns a list of the classes of `Star`s and `Galaxy`s on the known list that are usable in the current domain. The syntax is

```
knownlist
```

It is also possible to ask for a list of objects available in other domains; the command

```
knownlist DE
```

displays objects available in the `DE` (discrete event) domain.

Displaying Information on a the Current Galaxy or a Class

If invoked without an argument, the `print` command displays information on the current galaxy. If invoked with an argument, the argument is either the name of a `Star` (or `Galaxy`) contained in the current galaxy, or the name of a class on the known list, and information is shown about that `Star` (or `Galaxy`). The syntax is

`print`
`print` *star-name*
`print` *star-class*

3.2.3. Running the Simulation

Once a simulation has been constructed using the commands previously described (also see the `source` command in section 3.4.2), use the commands in this section to run the simulation.

Creating a Schedule

The `schedule` command generates and returns the schedule (the order in which `Star`s are invoked). For domains such as DE, this command returns a not-implemented message (since there is no "compile time" DE schedule as there is for SDF). Syntax:

```
schedule
```

Running the Simulation

The `run` command generates the schedule and runs it *n* times, where *n* is the argument (the argument may be omitted; its default value is 1). For the DE interpreter, this command runs the simulation for *n* time units, and *n* may be a floating point number (default 1.0). If this command is repeated, the simulation is started from the beginning. If animation is enabled, the full name of each star will be printed to the standard output when the star fires. Syntax:

`run`
`run` *n*

Continuing a Simulation

The **`cont`** command continues the simulation for *n* additional steps, or time units. If the argument is omitted, the default value of the argument is the value of the last argument given to a run or cont command (1.0 if no argument was ever given). The syntax is

```
cont
cont n
```

Wrapping Up a Simulation

The **`wrapup`** command calls the wrapup method of the current target (which, as a rule, will call the wrapup method of each **`Star`**), signaling the end of the simulation run. The syntax is

```
wrapup
```

Animating a Simulation

The **`animation`** command can be used to display on the standard output the name of each star as it runs. The syntax

```
animation on
```

enables animation, while

```
animation off
```

disables it. The syntax

```
animation
```

simply tells you whether animation is enabled or disabled.

3.2.4. Undoing what you've done

The commands in this section remove part or all of the structure you have built with previous commands.

Resetting the Interpreter

The **`reset`** command replaces **`mainGalaxy`** by an empty **`Galaxy`**. Any **`defgalaxy`** definitions you have made are still remembered. The syntax is

```
reset
```

Removing a Star

The `delstar` command removes the named Star from the current galaxy. The syntax is

```
delstar name
```

where *name* is the name of the star.

Removing a connection

The `disconnect` command reverses the effect of a previous `connect` or `nodeconnect` command. The syntax is

```
disconnect starname portname
```

where *starname* and *portname*, taken together, specify one of the two connected portholes. Note that you can disconnect by specifying either end of a porthole for a point-to-point connection.

Removing a node

The `delnode` command removes a node from the current galaxy. Syntax:

```
delnode node
```

3.2.5. Targets

Ptolemy uses a structure called a *target* to control the execution of a simulation, or, in code generation, to specify the target architecture for code generation. There is always a target; by default (if you issue no target commands), your target will have the name `default-XXX`, where `XXX` is replaced by the name of the current domain. Alternate targets for simulation can be used to specify different behavior (for example, to use a different scheduler). For code generation, the target contains information about the target of compilation, has methods for downloading code and starting execution, etc.

What Targets are Available?

The command

```
targets
```

returns the list of targets available for the current domain.

Changing the Target

The command

```
target
```

displays the target for the universe or current galaxy, together with its parameters. Specifying an argument, e.g.

`target` *new-target-name*

changes the target to *new-target-name*.

Changing Target Parameters

Target parameters may be changed with the **`targetparam`** command. The syntax is

`targetparam` *param-name* **"***new-value***"**

3.2.6. Miscellaneous Commands

This section describes the remaining interpreter commands.

Loading Commands from a File

For complicated simulations it is best to store your interpreter commands — at least those defining the simulation connectivity — in a file rather than typing them into the interpreter directly. This way you can run you favorite editor in one window, and run the interpreter from another window, easily modifying the simulation and also keeping a permanent record. An exception to this is simply changing **`States`** using the **`setstate`** command and running and continuing the simulation using **`run`** and **`cont`** — this is normally done interactively with the interpreter.

The **`source`** command reads interpreter commands from the named file, until end of file or a syntax error occurs. The # character indicates that the rest of the line is a comment. By convention, files meant to be read by the load command end in ".pt". Example:

```
source "testfile.pt"
```

The tilde notation for users' home directories is allowed; for example, try

```
source "~ptolemy/demo/ptcl/sdf/liss.pt"
```

It is also possible to specify a file to be loaded by the interpreter on the command line. If, when you start the interpreter you type

```
ptcl myCommands.pt
```

the interpreter will load the named file, execute its commands, and then quit. No commands will be prompted for.

The **`source`** command is actually built in to Tcl itself, but it is described here nevertheless.

Changing the seed of random number generation

The **`seed`** command changes the seed of the random number generation. The default value is 1. The syntax is

`seed` *n*

where *n* is an unsigned integer.

Changing the current directory

The `cd` command changes the interpreter's idea of the current directory. For example,

```
cd "~ptolemy/demo/ptcl/sdf"
source "liss.pt"
```

will load the same file as the example in the previous section. To see what the interpreter's current directory is, you can type

```
pwd
```

Dynamically linking in new stars

The interpreter has the ability to extend itself by linking in outside object files; the object files in question must define single stars (they will have the right format if they are produced from preprocessor input). Unlike Pigi, the graphical interface, the interpreter will not automatically run the preprocessor and compiler; it expects to be given object files that have already been compiled. The syntax is

`link "`*object-file-name***`"`**

Any star object files that are linked in must only call routines that are already linked into the interpreter. For that reason, it is possible that a star that can be linked into Pigi, the graphic interface, might not be linkable into the interpreter in some rare cases.

It is also possible to link in several object files, pull in functions from libraries, etc. by use of the `multilink` command. The syntax is

`multilink` *opt1 opt2 opt3 ...*

where the options may be the names of object files, linker options such as "-L" or "-l" switches, etc. These arguments are supplied to the Unix linker along with whatever options are needed to completely specify the incremental link.

When the above linker commands are used, the linked in code has temporary status; symbols for it are not entered into the symbol table (meaning that the code cannot be linked against by future incremental links), and it can be replaced: an error in the loaded modules could be corrected and the `link` or `multilink` command could be repeated. There is an alternative linking command that specifies that the new code is to be considered ‘‘permanent’’; it causes a new symbol table to be produced for use in future links. Such code cannot be replaced, but it can be linked against by future incremental link commands. The syntax is

`permlink` *opt1 opt2 opt3 ...*

where the options are the same as for the `multilink` command.

Top Blocks

The **`topblocks`** command, which takes no arguments, returns a list of the top-level blocks in the current galaxy.

Examining states

The **`statevalue`** command takes the form

```
statevalue block state
```

and returns the current value of the state *state* within the block *block*. The command takes an optional third argument, which may be either **`current`** to specify that the current value should be returned (the default), or **`initial`** to specify that the initial value should be returned.

Giving Up

The **`exit`** command exits the interpreter. The syntax is

```
exit
```

Getting Help

The **`help`** command implements a simple help system describing the commands available and their syntax. It does not provide help with the standard Tcl functions. The syntax is

```
help topic
```

or

```
help ?
```

for a list of topics. If the argument is omitted, a short "help on help" is printed.

3.2.7. Limitations of the Interpreter

There should be many more commands returning information on the simulation, to permit better exploitation of the full power of the Tcl language.

3.2.8. A Wormhole example

Here is an example of a simulation that contains both an SDF portion and a DE portion. In this example, a Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples. This demo shows how easy it is to use the SDF stars to perform computation on DE particles. The overall delay of the SDF wormhole is zero, so the result is simply Poisson arrivals of Gaussian noise samples.

A **`Wormhole`** has an *outer domain* and an *inner domain*. The outer domain is determined by the current domain at the time the user starts the **`defgalaxy`** command to create the wormhole. The inner domain is determined by the **`domain`** command that appears inside the galaxy definition.

```
# create the wormhole
domain DE
defgalaxy wormGuts {
     domain SDF
     star add Add; numports add input 2
     star IIDGaussian1 IIDGaussian
     alias out add output
     alias in add "input#1"
     connect IIDGaussian1 output add "input#2"
}
reset

# Creating the main universe.
domain DE
star wormGuts1 wormGuts
star Poisson1 Poisson; star graf XMgraph
numports graf input 2
setstate graf title "Noisy Poisson Process"
setstate graf options "-P -0 original -1 noisy"
node node1
nodeconnect Poisson1 output node1
nodeconnect wormGuts1 in node1
nodeconnect graf "input#1" node1
connect wormGuts1 out graf "input#2"

run 40
wrapup
```

4. Writing Stars

Authors:

Joseph Buck
Soonhoi Ha
Edward A. Lee

4.1. Introduction

Ptolemy provides rich libraries of stars for each domain. Since the stars in the libraries were designed to be as generic as possible, many complicated functions can be realized by a galaxy of those stars. Nonetheless, no star library can possibly be complete; you may need to design your own stars. For this purpose, the Ptolemy preprocessor language was devised. The main part of this chapter is devoted to the usage of the preprocessor language.

The newly designed stars can be dynamically linked into **Ptolemy** when they are required, which avoids the hassle of frequent recompilation of the system. If the new stars are generic and useful, however, it might be better to add them to the list of compiled-in stars and rebuild the system. This chapter explains what procedure should be followed to add or modify stars into the libraries.

4.2. The Ptolemy Processor Language

The Ptolemy preprocessor exists to make it easier to write and document star and galaxy class definitions to run under Ptolemy. Instead of writing all the initialization code required for a Ptolemy star, the user can concentrate on writing the action code for a star and let the preprocessor generate the standard initialization code for portholes, states, etc. The preprocessor also generates standardized documentation to be included in the manual.

The Ptolemy preprocessor reads class definitions from an input file, and generates one **`.cc,`** one **`.h`** and one **`.t`** file for each class defined. By convention, only one class is defined per file, and the filename is of the form **`MyClass.pl`** where **`MyClass`** is the name of the class to be defined (the dynamic linking code in Pigi searches for filenames of this form, so you might as well consider it mandatory to use this naming convention). The **`.t`** file is troff source for documentation.

The preprocessor is written in Yacc and C. It does not attempt to parse the parts of the language that consist of C++ code (for example, **`go()`** methods); for these it simply counts curly braces to find the ends of the items in question. It outputs **`#line`** directives so the C++ compiler will print error messages, if any, with respect to the original source file. It also makes no attempt to parse information in the **`explanation`** directive, so arbitrary troff formatting commands can be inserted.

To make things clearer, let's start with an example, a rectangular pulse star in the file **`SDFRect.pl`**:

```
ident {
/*  Anything put here will appear verbatim in the .cc and .h files
    so please note that comments must be delimited, as done here.
```

```
        This feature is no longer used by the Ptolemy group; its use
        is deprecated.
    */
    }
    defstar {
         name { Rect }
         domain { SDF }
         desc {
    Generates a rectangular pulse of height "height" (default 1.0).
    with width "width" (default 8).
         }
         version {%W% %G%}
         author { J. T. Buck }
         copyright { 1991 The Regents of the University of California }
         location { SDF main library }
         explanation {
    If needed, a longer explanation would go here.
         }
         defstate {
              name { height }
              type { float }
              default { 1.0 }
              desc { Height of the rectangular pulse. }
         }
         defstate {
              name { width }
              type { int }
              default { 8 }
              desc { Width of the rectangular pulse. }
         }
         defstate {
              name { count }
              type { int }
              default { 0 }
              desc { Internal counting state. }
              attributes { A_NONSETTABLE|A_NONCONSTANT }
         }
         output {        // the output port
              name { output }
              type { float }
              desc { The output pulse. }
         }
         go {            // the run-time function
              double t = 0.0;
              if (count < width) t = height;
              count = int(count) + 1;
              output%0 << t;
```

```
            }
        }
```

Running the preprocessor on the above file produces the three files **`SDFRect.h`**, **`SDFRect.cc`** and **`SDFRect.t`**; the names are determined *not* by the input filename but by concatenating the domain and name fields.

Three types of declarations may appear at the "top level" of a Ptolemy language file: an **`ident`** section, a **`defstar`** section, or a **`defgalaxy`** section. Any number of these may appear in the file, although normally, it is preferable to separate each star or galaxy definition into its own file. For each **`defstar`** or **`defgalaxy`** section, three files are created, a **`.h`** file declaring the class, a **`.cc`** file implementing its methods, and a **`.t`** file documenting it. The most recently seen **`ident`** section is inserted, uninterpreted, at the beginning of the **`.h`** and **`.cc`** files (the naming is historical; this section was originally used to generate identifying information in the output files, but it is no longer needed for this purpose). If it contains comments, they should be valid C++ comments. The **`defstar`** and **`defgalaxy`** sections are themselves composed of subitems that define various attributes of the star or galaxy. All subitems are of the form

keyword { *stuff* }

where the *stuff* may itself be composed of sub-sub-items, or may be C++ code (in which case the Ptolemy language processor checks it only for balanced curly braces). Note that the keywords are *not* reserved words; they may also be used as identifiers.

4.3. Using the preprocessor

The preprocessor permits many star definitions to be written in one file; processing the file results in one .cc file and one .h file for each class. However, this is not recommended. The recommended procedure (the *required* procedure if you want Pigi’s incremental linking and code viewing features to work correctly) is to write the definition for a star in domain XXX with name YYY in file XXXYYY.pl. Then, running the command

```
ptlang XXXYYY.pl
```

will produce the files XXXYYY.cc, XXXYYY.h, and XXXYYY.t. With this policy, the makefiles can contain the default rules

```
.pt.cc:
        ptlang $<

.pt.h:
        ptlang $<
```

See the ‘‘common Makefile rules’’ file ˜ptolemy/common.mk for how this can be made to work ‘‘in context’’.

The documentation file can be printed out using the command ˜ptolemy/doc/bin/printpage. By convention, the documentation file has the name XXXYYY.t, where XXX is the domain name and YYY is the star name. The printpage command assumes your PRINTER environment variable is set to the printer of choice. The following line in your .cshrc file, for example, will ensure this:

```
setenv PRINTER printer_name
```

4.4. Items that appear in a defstar

The following items can appear in a **`defstar`** directive. Some of these can also appear in a **`defgalaxy`** directive, as explained below.

name

This is a required item, and has the syntax

`name {` *identifier* **`}`**

It provides the name of the class to be defined, and (together with the domain) determines the names of the output files.

domain

This is a required item; it specifies the domain, such as SDF. Syntax:

`domain {` *identifier* **`}`**

where *identifier* specifies the domain. Currently, the SDF, DDF, and DE domains are fully supported; for the Thor domain, a separate preprocessor is normally used.

derivedfrom

This optional item indicates that the star is derived from another class. Syntax:

`derivedfrom {` *identifier* **`}`**

where *identifier* specifies the base class. The **`.h`** file for the base class is automatically included in the output **`.h`** file.

An important thing to remember for derived stars is that the real C++ classname is formed by concatenating the user-visible name and the domain name. Thus the C++ classname for the star **`LMS`** in the **`SDF`** domain is **`SDFLMS`**. The **`derivedfrom`** statement, which may also be written **`derivedFrom`** or **`derived`**, allows you to say either

```
derivedfrom { FIR }
```

or

```
derivedfrom { SDFFIR }
```

to specify the base class.

descriptor

This item defines the descriptor for the class. It has the syntax

`descriptor {` *text* **`}`**

or

`desc {` *text* **`}`**

where *text* is simply a section of text that will become the short descriptor of the star or galaxy. A principal use of the short descriptor is to get on-screen help, so the descriptor should not include any troff formatting commands. Unlike the **`explanation`** (described below), it does not pass through troff. The following are legal descriptors:

```
     desc { A one line descriptor. }
     desc {
A multi-line descriptor.
The same line breaks and spacing will be used when the descriptor
is displayed on the screen.
     }
```

By convention, in these descriptors, references to the names of states, inputs, and outputs should be enclosed in quotation marks. Also, each descriptor should begin with a capital letter, and end with a period. If the descriptor seems to get long, use the **`explanation`** directive, explained below.

version

This item contains two entries as shown below

`version {` *number MO/DA/YR* **`}`**

where the number is a version number, and the MO/DA/YR is the version date. If you are using SCCS for version control then the following syntax will work well:

```
version { %W% %G% }
```

When the file is checked in by SCCS, the string "%W%" will be replaced with a string of the form: "@(#)filename num", where num is the version number, and "%G%" will be replaced with a properly formatted date.

author

This optional entry identifies the author of the star or galaxy, and may contain any number of names. The syntax is

`author {` *first name, second name and third name* **`}`**

Any set of characters between the brackets will be interpreted as a list of author names.

acknowledge

This optional entry attaches an acknowledgement section to the documentation. The syntax is

`acknowledge {` *arbitrary single line of text* **`}`**

copyright

This optional entry attaches a copyright notice to the **`.h`**, **`.cc`**, and **`.t`** files. The syntax is

`copyright {` *copyright information* **`}`**

For example, we used to use the following (our lawyers have recently caused us to increase the verbosity):

```
copyright { 1991 The Regents of the University of California }
```

location

This item describes the location of a star definition. The following descriptions are used, for example:

```
location { SDF main library }
```

or

`location {` *directory* **`}`**

explanation

This item is used to give longer explanations of the function of the stars. The explanation will appear only in the **`.t`** documentation file. The syntax is

`explanation {`
body
`}`

The body is assumed to be troff source, but a number of macros are provided for convenience.

The most frequently used macro in **`explanation`** sections is the un-numbered header, invoked as follows:

```
.UH "Header title"
```

Star portHole and state names, by convention, should be given in *italics*. This can be conveniently done using the font-change escapes, as in "\fIitalics\fP". **Ptolemy** class names are, by convention, set in a special font. This font may be obtained in-line by the escape sequence "\f(*F", and again the previous font can be restored using "\fP". Alternatively, the font can be obtained using the troff macro:

```
.c word
```

or the start and end macros

```
.(c
body
.)c
```

defstate

This item is used to define a state; it has other items inside it. Here is an example of a state definition:

```
defstate {
     name { gain }
     type { int }
     default { 1.0 }
     desc { Output gain. }
     attributes { A_CONSTANT|A_SETTABLE }
}
```

There are five types of sub-items that may appear in a defstate statement, in any order. The **`name`** field is the name of the state; the **`type`** field is its type, which may be one of **`int`**, **`float`**, **`string`**, **`complex`**, **`intarray`**, **`floatarray`**, **`complexarray`**, or **`stringarray`**. Case is ignored for the type argument.

The **`default`** item specifies the default value of the state; its argument is either a string (enclosed in quotation marks) or a numeric value. The above entry could equivalently have been written:

```
default { "1.0" }
```

Furthermore, if a particularly long default is required, as for example when initializing an array, the string can be broken into a sequence of strings. The following example shows the default for a ComplexArray:

```
default {
"(-.040609,0.0) (-.001628,0.0) (.17853,0.0) (.37665,0.0)"
"(.37665,0.0) (.17853,0.0) (-.001628,0.0) (-.040609,0.0)"
}
```

The **`desc`** (or **`descriptor`**) item, which is optional but highly recommended, attaches a descriptor to the state. The same formatting options are available as with the star descriptor.

Finally, the **`attributes`** keyword specifies state attributes. At present, two attributes are defined for all stars: A_CONSTANT and A_SETTABLE (along with their complements A_NONCONSTANT and A_NONSETTABLE). If a state has the A_CONSTANT attribute, then its value is not modified by the run-time code in the star (it is up to you as the star writer to ensure that this condition is satisfied). States with the A_NONCONSTANT attribute may change when the star is run. If a state has the A_SETTABLE attribute, then user interfaces (such as *pigi*) will prompt the user for values when directives such as *edit-parameters* are given. States without this attribute are not presented to the user; such states always start with their default values as the initial value. If no attributes are specified, the default is A_CONSTANT|A_SETTABLE.

Code generation stars use a great number of attributes, many specific to the model for which code is being generated. Read the documentation for the appropriate code generation domain to learn more about these.

Attributes are currently ignored for galaxy states.

Mechanisms for accessing and updating states in C++ methods associated with a star are explained below, in the section entitled ‘‘Writing C++ code.’’

input, output, inmulti, outmulti

These keywords are used to define a porthole, which may be an input or output porthole or an input or output multiporthole. Like **`defstate`**, it contains other items. Here is an example:

```
input {
      name { signalIn }
      type { complex }
      numtokens { 2 }
      desc { A complex input that consumes 2 input particles. }
}
```

Here, **`name`** specifies the porthole name. This is a required item. **`type`** specifies the particle type. The type may be one of **`int`**, **`float`**, **`complex`**, **`message`**, or **`anytype`**. Again, case does not matter for the type value. The **`type`** item may be omitted; the default type is **`anytype`**. For more information on **`message`**, please see the appendix at the end of this section.

The **`numtokens`** keyword (it may also be written **`num`** or **`numTokens`**) specifies the number of tokens consumed or produced on each firing of the star; if the item is omitted, a value of one is used. For stars where this number depends on the value of a state, it is preferable to leave out the **`numtokens`** specification and to have the **`start`** method set the number of tokens (in the SDF domain and most code generation domains, this is accomplished with the **`setSDFParams`** method). This item is used primarily in the SDF and code generation domains, and is discussed further in the documentation of those domains.

There is an alternative syntax for the type field of a porthole; this syntax is used in connection with ANYTYPE to specify a connection between the types of two portholes. The syntax is

`type { =` *name* **`}`**

where *name* is the name of another porthole. This indicates that this porthole inherits its type from the specified porthole. For example, here is the definition of the SDF Fork star:

```
defstar {
     name {Fork}
     domain {SDF}
     desc { A fork function: copies input particles to each output. }
     input {
          name{input}
          type{ANYTYPE}
     }
     version {2.10 12/8/92}
     outmulti {
          name{output}
          type{= input}
          desc{ Type is inherited from the input. }
     }
     go {
          MPHIter nextp(output);
          PortHole* p;
          while ((p = nextp++) != 0)
               (*p)%0 = input%0;
     }
}
```

Full details for accessing portholes C++ methods associated with a star are explained below, in the section entitled ‘‘Writing C++ code for stars.’’

constructor

This item allows the user to specify extra C++ code to be executed in the constructor for the class. This code will be executed *after* any automatically generated code to initialize portholes, states, etc. Syntax:

constructor { *body* **}**

where *body* is a piece of C++ code. It can be of any length. Note that actions that must be performed before every simulation run should appear in the start method, not the constructor.

conscalls

You may want to have members in your star or galaxy that have constructors that require arguments. These members would be added by using the **`public`**, **`private`**, or **`protected`** keywords. If you have such members, the **`conscalls`** keyword provides a mechanism for passing arguments to the constructors. Simply list the names of the members followed by the list of constructor arguments for each, separated by commas if there is more than one. Syntax:

`conscalls {` *member1(arglist), member2(arglist)* **`}`**

Note that *member1,* and *member2 should have been previously defined in a* **`public`**, **`private`***, or* **`protected`** *section.*

destructor

This item defines a destructor. Syntax:

`destructor {` *body* **`}`**

You generally need a destructor only if you allocate memory in the constructor; termination functions that happen with every run should appear in the **`wrapup`** function. The optional keyword **`inline`** may appear before **`destructor`**; if so, the destructor function definition appears inline, in the header file. Since the destructor for all stars is virtual, this is only a win when the star is used as a base for derivation.

start

This item defines the **`setup()`** method, which is called every time the simulation is restarted. Syntax:

`setup {` *body* **`}`**

The optional keyword **`inline`** may appear before the **`setup`** keyword. It is common for this method to set parameters of input and output portholes, and to initialize states. The code syntax for doing this is explained below.

The alternative keyword **`start`** is accepted instead of **`setup`** for backward compatibility with previous releases.

go

This item defines the action taken by the star when it is fired. Syntax:

`go {` *body* **`}`**

The optional keyword **`inline`** may appear before the **`go`** keyword. The go methods will typically read input particles and write outputs. The code syntax for doing this is explained below.

wrapup

This item defines the **`wrapup()`** method, which is called at the completion of a simulation. Syntax:

`wrapup {` *body* **`}`**

The optional keyword **`inline`** may appear before the **`wrapup`** keyword. The wrapup methods might display or store final state values. The code syntax for doing this is explained below.

exectime This item defines the optional `myExecTime()` function, which is used in code generation to specify how many time units are required to execute the star's code. The item is very similar to the `go` item, except that the function must return an integer value. Syntax:

```
exectime { body }
```

The optional keyword `inline` may appear before the `exectime` keyword.

codeblock

Codeblocks are parametized blocks of code for use in code generation stars. Their use and format is discussed in detail in the documentation for code generation domains. Syntax:

```
codeblock {
     code
     ...
}
```

public, protected, private

These three keywords allow the user to declare extra members for the class with the desired protection. Syntax:

protkey body

where *protkey* is `public`, `protected`, or `private`. Example, from the Xgraph star:

```
protected {
     FILE *strm;
     int delFile;
     int index;
     const char* fileName;
}
```

If any of the added members require arguments for their constructors, use the `conscalls` item to specify them.

ccinclude, hinclude

These directives cause the `.cc` file, or the `.h` file, to `#include` extra files. By default, the `.cc` file includes `KnownBlock.h` and the generated `.h` file; the `.h` file will (for stars) include *XXX*`Star.h`, where *XXX* is the domain name, or (for galaxies) `Galaxy.h`, as well as the files for any states that are defined, and if a `derived` section is found, the name of the baseclass as well. Syntax:

```
ccinclude { inclist }
hinclude { inclist }
```

where *inclist* is a comma-separated list of include files. Each filename must be surrounded either by quotation marks or by "<" and ">" (for system include files like **`<math.h>`**).

code

This keyword allows the user to specify a chunk of arbitrary C++ code. This code is inserted into the **`.cc`** file after the include files but before everything else; it can be used to define static non-class functions, declare external variables, or anything else. The outermost pair of curly braces is stripped. Syntax:

`code {` *body* **`}`**

header

This keyword allows the user to specify an arbitrary set of definitions that will appear in the header file. Everything between the curly brackets is inserted into the **`.h`** file after the include files but before everything else. The outermost pair of curly braces is skipped.

method

The **`method`** item provides a fully general way to specify an additional method for the class of star or galaxy that is being defined. Here is an example:

```
virtual method {
     name { exec }
     access { protected }
     arglist { "(const char* extraOpts)" }
     type { void }
     code {
          // code for the SDFXgraph::exec method goes here
     }
}
```

An optional function type specification may appear before the **`method`** keyword, which must be one of the following:

```
virtual
inline
pure
pure virtual
inline virtual
```

The **`virtual`** keyword makes a virtual member function. If the **`pure virtual`** keyword is given, a pure virtual member function is declared and there must be no **`code`** item in this case. The function type **`pure`** is a synonym for **`pure virtual`**. The **`inline`** function type declares the function to be inline.

Here are the **`method`** subitems:

`name`:

The name of the method. This is a required item.

`access`:

The level of access for the method, one of **`public`**, **`protected`**, or **`private`**. If the item is omitted, **`protected`** is assumed.

`arglist`:

The argument list for the method, as a quoted string. If omitted, the method has no arguments.

`type`:

The return type of the method. If the return type is not a single identifier, you must put quotes around it. If it is omitted, the return type is **`void`** (no value is returned).

`code`:

The code that implements the method. This is a required item, unless the **`pure`** keyword appears, in which case this item *cannot* appear.

4.5. Items that appear in galaxies

Items that are the same as in defstar

The following items, described above, take exactly the same form in a **`defgalaxy`** as in a **`defstar,`** and hence need not be discussed further:

```
name { }
descriptor { }
version { }
author { }
acknowledge { }
copyright { }
location { }
explanation { }
defstate { }
constructor { }
conscalls { }
destructor { }
start { }
wrapup { }
public { }
protected { }
private { }
ccinclude { }
hinclude { }
code { }
method { }
```

Other directives are necessary for constructing a galaxy, but these features have not yet been implemented, so they cannot be discussed.

4.6. Writing C++ code for stars

This section assumes a knowledge of the C++ language; no attempt will be made to teach the language. We recommend ‘‘C++ Primer’’, by Stanley Lippman (from Prentice-Hall) for those new to the language.

C++ code segments are an important part of any star definition. They can appear in the **`setup`**, **`go`**, **`wrapup`**, **`constructor`**, **`destructor`**, **`exectime`**, **`code`**, and **`method`** directives in the **Ptolemy** preprocessor. These directives all include a body of arbitrary C++ code, enclosed by curly braces, ‘‘{’’ and ‘‘}’’. In all but the **`code`** directive, the C++ code between braces defines the body of a method of the star class. This method can access any member of the class, including portholes (for input and output), states, and members defined with the **`public`**, **`protected`**, and **`private`** directives.

4.6.1. The structure of a Ptolemy star

In general, the task of a Ptolemy star is to receive input Particles and/or produce output Particles; in addition, there may be side effects (reading or writing files; displaying graphs). As for all C++ objects, the constructor is called when the object is created, and the destructor is called when the object is destroyed. In addition, the setup() method, if any, is called every time a new simulation run is started, the go() method (which always exists except for stars like BlackHole and Null that do nothing) is called each time a star is to be executed, and the wrapup() method is called to complete the simulation run.

4.6.2. Reading inputs and writing outputs

The precise mechanism for references to input and output portholes depends somewhat on the domain. This is because stars in the domain XXX use objects of class InXXXPort and OutXXXPort (derived from PortHole) for input and output, respectively. We will explain here how such references are used in the SDF domain. A later section discusses the DE domain. For information about other domains, please see the domain document in the Almagest.

PortHoles and Particles

In the SDF domain, normal inputs and outputs become members of type **`InSDFPort`** and **`OutSDFPort`** after the preprocessor is finished. These are derived from base class **`PortHole`**. For example, given the following directive in the **`defstar`** of an SDF star,

```
input {
      name {in}
      type {float}
}
```

a member named **`in`**, of type **`InSDFPort`**, will become part of the star.

We are not usually interested in directly accessing these porthole classes, but rather wish to read or write data through the portholes. All data passing through a porthole is derived from base class **`Particle`**. Each particle contains data of the type specified in the **`type`** subdirective of the **`input`** or **`output`** directive. Currently this can be float, int, complex, or ‘‘message’’.

The operator ‘‘%’’ operating on an SDF porthole returns a reference to a particle. Consider the following example:

```
go {
      Particle& currentSample = in%0;
      Particle& pastSample = in%1;
      ...
}
```

The right-hand argument to the ‘‘%’’ operator specifies the delay of the access. A zero always means the most recent particle. A one means the particle arriving just before the most recent particle. The same rules apply to outputs. Given an output named ‘‘out’’, the same particles that are read from ‘‘in’’ can be written to ‘‘out’’ in the same order as follows:

```
go {
      ...
      out%1 = pastSample;
      out%0 = currentSample;
}
```

This works because `out%n` returns a *reference* to a particle, and hence can accept an assignment. The assignment operator for the class `Particle` is overloaded to make a copy of the data field of the particle.

Operating directly on class `Particle`, as in the above examples, is useful for writing stars that accept ANYTYPE of input. The operations need not concern themselves with the type of data contained by the particle. But it is far more common to operate numerically on the data carried by a particle. This can be done using a cast to a compatible type. For example, since ‘‘in’’ above is of type `float`, its data can be accessed as follows:

```
go {
      Particle& currentSample = in%0;
      double value = double(currentSample);
      ...
}
```

or more concisely,

```
go {
      double value = double(in%0);
      ...
}
```

The expression `double(in%0)` can be used anywhere that a double can be used. In many contexts, where there is no ambiguity, the conversion operator can be omitted:

```
      double value = in%0;
```

However, since conversion operators are defined to convert Particles to several types, it is often necessary to indicate precisely which type conversion is desired.

To write data to an output porthole, note that the right-hand side of the assignment operator should be of type **`Particle`**, as shown in the above example. An operator ‘‘<<’’ is defined for particle classes to make this more convenient. Consider the following example:

```
go {
     float t;
     t = some value to be sent to the output
     out%0 << t;
}
```

Note the distinction between the ‘‘<<’’ operator and the assignment operator; the latter operator copies Particles, the former operator loads data into particles. The type of the right-side operand of ‘‘<<’’ may be **`int`**, **`float`**, **`double`**, **`Complex`** or **`Envelope;`** the appropriate type conversion will be performed. For more information on the **`Envelope`** and **`Message`** types, please see the appendix at the end of this section.

SDF PortHole parameters

In the above example, where **`in%1`** was referenced, some special action is required to tell **Ptolemy** that past input particles are to be saved. Special action is also required to tell the SDF scheduler how many particles will be consumed at each input and produced at each output when a star fires. This information can be provided through a call to **`setSDFParams`** in the **`setup`** method. This has the syntax

```
setup {
     name.setSDFParams(multiplicity, past)
}
```

where *name* is the name of the input or output porthole, *multiplicity* is the number of particles consumed or produced, and *past* is the number of past samples accessed. For the above example, *past* would have to be at least one.

Multiple PortHoles

Sometimes a star should be defined with *n* input portholes or *n* output portholes, where *n* is variable. This is supported by the class **`MultiPortHole`** and its derived classes. An object of this class is a sequential list of **`PortHole`**s. For SDF, we have the specialized derived class **`MultiInSDFPort`** (which contains **`InSDFPorts`**) and **`MultiOutSDFPort`** (which contains **`OutSDFPorts`**).

Defining a multiple porthole is easy, as illustrated below:

```
defstar {
     ...
     inmulti {
          name {input_name}
          type {input_type}
     }
     outmulti {
          name {output_name}
          type {output_type}
     }
}
```

To successively access individual portholes in a **`MultiPortHole`**, the **`MPHIter`** iterator class should be used. Consider the following example:

```
defstar {
     name {Fork}
     domain {SDF}
     desc { Copies input particles to each output. }
     input {
          name{input}
          type{ANYTYPE}
     }
     outmulti {
          name{output}
          type{= input}
     }
     go {
          MPHIter nextp(output);
          PortHole* p;
          while ((p = nextp++) != 0)
               (*p)%0 = input%0;
     }
}
```

A single input porthole supplies a particle that gets copied to any number of output portholes. The **`type`** of the **`output MultiPortHole`** is inherited from the type of the input. The first line of the **`go`** method creates an **`MPHIter`** iterator called **`nextp`**, initialized to point to portholes in **`output`**. The ‘‘++’’ operator on the iterator returns a pointer to the next porthole in the list, until there are no more portholes, at which time it returns zero. So the **`while`** construct steps through all output portholes, copying the input particle data to each one.

Consider another example, which sums any number of inputs:

```
defstar {
     name {Add}
     domain {SDF}
     desc { Output the sum of the inputs, as a floating value.  }
     inmulti {
          name {input}
          type {float}
     }
     output {
          name {output}
          type {float}
     }
     go {
          MPHIter nexti(input);
          PortHole *p;
          double sum = 0.0;
          while ((p = nexti++) != 0)
               sum += double((*p)%0);
          output%0 << sum;
     }
}
```

Again, an **`MPHIter`** iterator named **`nexti`** is created and used to access the inputs.

Upon occation, the **`numberPorts()`** method of class **`MultiPortHole`**, which returns the number of ports, is useful.

Type conversion

The type conversion operators and ‘‘<<’’ operators are defined as virtual methods in the base class **`Particle`**. There are never really objects of class **`Particle`** in the system; instead, there are objects of class **`IntParticle`**, **`FloatParticle`**, **`ComplexParticle`**, which hold data of type **`int`**, **`double`** (not float!), and **`Complex`**, respectively (there is also **`MessageParticle`**, which is described later). The conversion and loading operators are designed to ‘‘do the right thing’’ when an attempt is made to convert between mismatched types.

Clearly we can convert an **`int`** to a **`double`** or **`Complex`**, or a **`double`** to a **`Complex`**, with no loss of information. Attempts to convert in the opposite direction work as follows: conversion of a **`Complex`** to a **`double`** produces the magnitude of the complex number. Conversion of a **`double`** to an **`int`** produces the greatest integer that is less than or equal to the **`double`** value. There are also operators to convert to or from **`float`**.

This gives considerable flexibility in defining code that operates on ANYTYPE. For instance, since ANYTYPE of particle can be cast to double, a plotting star can accept ANYTYPE of input, and internally the star only has to cast the input to double. Of course, if the input is complex, some information is lost; thus when complex values are fed directly to the Xgraph star, the magnitudes of the values are plotted.

Each particle also has a virtual **`print()`** method, so a star that writes particles to a file can also accept ANYTYPE.

PortHoles in the DE domain

For convenience, we give here a brief description of how data is accessed in the DE domain. More details are given in the DE domain section of the Almagest.

In the DE domain, there is a time associated with each event; this time is set in the star's **`arrivalTime`** member. When a star's **`go()`** function is called, the **`arrivalTime`** member contains the current time. If there is only one input **`DEPortHole`**, then it is guaranteed that there is a Particle in that **`DEPortHole`**. If the star has more than one input porthole, then you must poll to find out which porthole has new data; that is done by checking the **`dataNew`** member of the **`DEPortHole`**. It is possible for simultaneous events to arrive on several different input portholes. However, if simultaneous events arrive on the same input porthole, the scheduler fires the star two separate times, presenting the events one at a time. The result is that any input porthole has at most one event (particle).

For convenience a **`completionTime`** data member is provided in **`DEStar`**s, but the user need not use it; it is initialized to zero at the start of execution. By convention, **`completionTime`** is used to compute the timestamp to be used for output particles; doing this in a data member allows it to be remembered across executions. However, the star writer is not required to use **`completionTime`** at all.

Particles may be read from **`DEPortHole`**s either by using **`input%0`** as for other types of portholes, or by using the **`get()`** method. **`input.get()`** is identical to **`input%0`** except that the **`dataNew`** flag of input is cleared as well. To write Particles to output DEPortHoles, the **`put(double)`** method is used. The argument to **`put`** is a timestamp value, and it returns a reference to a Particle; you may copy another Particle to this reference or load data into it. For example:

```
output.put(completionTime) = input.get();

output.put(completionTime) << dataValue;
```

If a particle arrives on an input and you do not **`get()`** it, it is discarded; it is not queued (to implement a queue, the **`Queue`** class defined in **`DataStruct.h`** may be used). In many cases an input signal is used for timing control (as a clock input) and you don't care about the data value at all; in that case, code typically just resets the **`dataNew`** flag of the porthole. A **`DEStar`** must have at least one input porthole to work correctly. How, then, are source stars defined? They are produced by deriving from class **`DERepeatStar`**. A **`DERepeatStar`** has a hidden pair of portholes that form a feedback loop, but this is hidden from you and you don't need to worry about it. A **`DERepeatStar`** is always fired at time zero, and the method **`refireAtTime(double)`** is provided; this allows the star to schedule itself for later execution. In many cases it is convenient to schedule the first execution at time zero; think of this as actually occurring at time epsilon for a very small epsilon. If a **`DERepeatStar`** defines a setup function, that function must call **`DERepeatStar::setup()`**.

Here is a (simplified) version of DEClock, which outputs a zero-valued token every 5 time units (the real star is parametized):

```
go {
     output.put(completionTime) << 0.0;
     refireAtTime(completionTime);
     completionTime += 5.0;
}
```

Notice that there are two firings at time zero; the first produces a token at time zero, the second at time 5.0.

When MultiPortHoles are used, we can't just use an `MPHIter` because it returns a pointer to class `PortHole` and we need to use `DEPortHole` specific methods. To handle this, two additional iterator classes are provided: `OutDEMPHIter`, for output multiportholes, and `InDEMPHIter`, for input multiportholes. The former returns a pointer to `OutDEPort` on each execution, and the latter returns a pointer to `InDEPort` on each execution.

Here is the `go()` function from `DEFork`:

```
go {
     completionTime = arrivalTime;
     Particle& pp = input.get();
     OutDEMPHIter nextp(output);
     OutDEPort *oport;
     while ((oport = nextp++) != 0)
          oport->put(completionTime) = pp;
}
```

Here is the go() function from `DEMerge`. Here, we use the dataNew flag to see which input(s) contain(s) new data:

```
go {
     completionTime = arrivalTime;
     InDEMPHIter nextp(input);
     InDEPort* iport;
     while ((iport = nextp++) != 0) {
          if (iport->dataNew)
               output.put(completionTime) = iport->get();
     }
}
```

4.6.3. States

This section is not domain-specific; states are the same in any domain. A state is defined by the `defstate` directive. The star can use a state to store data values, remembering them from one invocation to another. They differ from ordinary members of the star, which are defined using the `public`, `protected`, and `private` directives, in that they have a name, and can be accessed from outside the star in systematic ways. For instance, the graphical interface Pigi permits the user to set any state with the A_SETTABLE attribute to some value prior to a run, using the *edit-params* command. The interpreter provides similar functionality through the `setstate` command. The state attributes are set in the `defstate` directive. A state may be modified by the star code during a run. The attribute A_NONCONSTANT is used as a pragma to mark a state as one that gets modified during a run. There is currently no mechanism for checking the correctness of these attributes.

All states are derived from the base class **`State`**, defined in the **Ptolemy** kernel. The derived state classes currently defined in the kernel are **`FloatState`**, **`IntState`**, **`ComplexState`**, **`StringState`**, **`FloatArrayState`**, **`IntArrayState`**, **`ComplexArrayState`**, and **`StringArrayState`**.

A state can be used in a star method just like the corresponding predefined data types. As an example, suppose the star definition contains the following directive:

```
defstate {
      name { myState }
      type { float }
      default { 1.0 }
      descriptor { Gain parameter. }
}
```

This will define a member of class **`FloatState`** with default value 1.0. No attributes are defined, so A_CONSTANT and A_SETTABLE, the default attributes, are assumed. To use the value of a state, it should be cast to type **`double`**, either explicitly by the programmer or implicitly by the context. For example, the value of this state can be accessed in the **`go`** method as follows:

```
go {
      output%0 << myState * double(input%0);
}
```

The references to input and output are explained above. The reference to **`myState`** works because an unambiguous cast to **`double`** is defined in **`FloatState`** class, and it is automatically invoked by the C++ compiler. (Similarly, a cast to **`int`** is defined for **`IntState`**, to **`Complex`** from **`ComplexState`**, and to **`char*`** for **`Stringstate`**).

Note that the type **`Complex`** is not a fundamental part of C++. We have implemented a subset of the **`Complex`** class as defined by several library vendors; we use our own version for maximum portability. Using the **`ComplexState`** class will automatically ensure the inclusion of the appropriate header files. A member of the **`Complex`** class can be initialized and operated upon any number of ways. For details, see the file ˜ptolemy/src/kernel/ComplexSubset.h.

A state may be updated by ordinary assignment in C++, as in the following lines

```
double t;
t = some value;
myState = t;
```

This works because the **`FloatState`** class definition has overloaded the assignment operator (‘‘=’’) to set its value from a double. Similarly, an **`IntState`** can be set from an **`int`** and a **`StringState`** can be set from a **`char*`** or **`const char*`**.

4.6.4. Array States

The **`ArrayState class`**es (**`FloatArrayState`**, **`IntArrayState`** and **`ComplexArrayState`**) are used to store arrays of data. For example,

```
defstate {
     name { taps }
     type { FloatArray }
     default { "0.0 0.0 0.0 0.0" }
     descriptor { An array of length four. }
}
```

defines an array of type **`double`** with dimension four, with each element initialized to zero. Quotes must surround the initial values. Alternatively, you can specify a file name with a prefix **`<`**. If you have a file named "foo" that contains the default values for an array state, you can write,

```
default { "< foo" }
```

The format of the file is also a sequence of data separated by spaces (or newlines, tabs, or commas). File input can be combined with direct data input as in

```
default { "<foo 2.0" }
default { "0.5 <foo <bar" }
```

A "repeat" notation is also supported for **`ArrayState`** objects: the two value strings

```
default { "1.0 [5]" }
default { "1.0 1.0 1.0 1.0 1.0" }
```

are equivalent. Any integer expression may appear inside the braces.

The number of elements in an **`ArrayState`** can be determined by calling its **`size()`** method. The size is not specified explicitly, but is calculated by scanning the default value.

As an example of how to access the elements of an **`ArrayState`**, suppose **`fState`** is a **`FloatState`** and **`aState`** is a **`FloatArrayState`**. The accesses, like those in the following lines, are routine:

```
sState = aState[1] + 0.5;
aState[1] = (double)sState * 10.0;
aState[0] = sState * aState[2];
```

For a more complete example of the use of **`FloatArrayState`**, consider the class **`FIR`** defined below. Note that this is a simplified version of the SDF **`FIR`** star that does not permit interpolation or decimation.

```
defstar {
     name {FIR}
     domain {SDF}
     desc {
A Finite Impulse Response (FIR) filter.
     }
     version {@(#)code     1.8 12/8/92}
     author { E. A. Lee }
     input {
          name {signalIn}
          type {float}
     }
     output {
          name {signalOut}
          type {float}
     }
     defstate {
          name {taps}
          type {floatarray}
          default {
     "-.040609 -.001628 .17853 .37665 .37665 .17853 -.001628 -.040609"
          }
          desc { Filter tap values. }
     }
     setup {
          // tell the PortHole the maximum delay we will use
          signalIn.setSDFParams(1, taps.size() - 1);
     }
     go {
          double out = 0.0;
          for (int i = 0; i < taps.size(); i++)
               out += taps[i] * double(signalIn%i);
          signalOut%0 << out;
     }
}
```

Notice the **`setup`** method; this is necessary to allocate a buffer in the input **`PortHole`** large enough to hold the particles that are accessed in the **`go()`** method. Notice the use of the **`size()`** method of the **`FloatArrayState.`**

We now illustrate an interpreter session using the above **`FIR Star`**. Assume there is a galaxy called **`SinGen`** that generates a sine wave. you can use it with the **`FIR star,`** as in:

```
galaxy foop SinGen
star fir FIR
star printer Printer
connect foop output fir signalIn
connect fir signalOut printer input
print fir
Star: mainGalaxy.fir
     ...
States in the star fir:
mainGalaxy.fir.taps type: FloatArray
 initial value: -.040609 -.001628 .17853 .37665 .37665 .17853 -.001628 -.040
 current value:
0 -0.040609
1 -0.001628
2 .17853
3 .37665
4 .37665
5 .17853
6 -0.001628
7 -0.040609
```

Then you can re-define taps by reading them from a file "foo", which contains the data:

```
1.1
-2.2
3.3
-4.4
```

The resulting interpreter commands are:

```
setstate fir taps "<foo 5.5"
print fir
Star: mainGalaxy.fir
     ...
States in the star fir:
mainGalaxy.fir.taps type: FloatArray
 initial value: <foo 5.5
 current value:
0 1.1
1 -2.2
2 3.3
3 -4.4
4 5.5
PTOLEMY:
```

This illustrates that *both* the contents and the size of a **`FloatArrayState`** are changed by a **`setstate`** command. Also, notice that file values may be combined with string values; when

```
< filename
```

occurs in an *initial-value*, it is processed exactly as if the whole file were substituted at that point.

Modifying states in derived classes

When one star is derived from another, it inherits all the states of the baseclass star. Sometimes we want to modify some aspect of the behavior of a baseclass state in the derived class. This is done by placing calls to member functions of the state in the constructor of the derived star. Useful functions include `setInitValue` to change the default value, and `setAttibutes` and `clearAttributes` to modify attributes.

4.6.5. Using random numbers

Ptolemy uses the GNU library routines for the random number generation. Refer to Volume II of the Art of Computer Programming by Knuth for details about the method. There are built-in classes for some popular distributions: uniform, exponential, geometric, discrete uniform, normal, log-normal, and so on. These classes use a common basic random number generation routine which is realized in the **ACG** class. Here are some examples of using random numbers.

The first example is the part of the DE Poisson star preprocessor file.

```
            hinclude { <NegExp.h> }
            ccinclude { <ACG.h> }
            protected {
                    NegativeExpntl *random;
            }
    // declare the static random-number generator in the .cc file
            code {
                    extern ACG* gen;
            }
            constructor {
                    random = NULL;
            }
            destructor {
                    if(random) delete random;
            }
            setup {
                    if(random) delete random;
                    random = new NegativeExpntl(double(meanTime),gen);
                    DERepeatStar :: setup();
            }
            go {
             ......
               // Generate an exponential random variable.
               double p = (*random)();
             ......
          }
```

The built-in class for an exponentially distributed random numbers is **NegativeExpntl**.

The Ptolemy kernel provides a single object to generate a stream of random numbers; the global variable *gen* (a poor choice of name, perhaps, but it's there now) is a pointer to it. We define a member of **NegativeExpntl** class and create an instance in the **setup()** method, not in the constructor since Ptolemy allows you to change the seed of the random number. When the user changes the seed of the random number generator, the object pointed to by *gen* is deleted and re-created, so all objects such as the one pointed to by *random* in this star become invalid.

Finally, we can read a random number of the specific type by calling operator **()** of the **NegativeExpnl** class.

This example uses a uniformly distributed random number.

```
            hinclude { <Uniform.h> }
            ccinclude { <ACG.h> }
            protected {
                    Uniform *random;
            }
    // declare the extern random-number generator in the .cc file
            code {
                    extern ACG* gen;
            }
            constructor {
                    random = NULL;
            }
            destructor {
                    if(random) delete random;
            }
            setup {
                    if(random) delete random;
                    random = new Uniform(0,double(output.numberPorts()),gen);
            }

            go {
               ......
               double p = (*random)();
             ......
          }
```

You may notice that the two examples above are very similar in nature. You can get another kind of distribution for the random numbers, by including the appropriate library file in .h file and by creating the instance with the right parameters in the **setup()** method.

4.6.6. Programming examples

The following star has no inputs, just an output. It generates a linearly increasing or decreasing sequence of float particles on its output:

```
defstar {
     name { FloatRamp }
     domain { SDF }
     desc {
Generates a ramp signal, starting at "value" (default 0)
with step size "step" (default 1).
     }
     output {
          name { output }
          type { float }
     }
     defstate {
          name { step }
          type { float }
          default { 1.0 }
          desc { Increment from one sample to the next. }
     }
     defstate {
          name { value }
          type { float }
          default { 0.0 }
          desc { Initial (or latest) value output by Ramp. }
          attributes { A_SETTABLE|A_NONCONSTANT }
     }
     go {
          double t = value;
          output%0 << t;
          t += step;
          value = t;
     }
}
```

The state **`value`** is initialized to define the value of the first output. Each time the star **`go()`** method fires, the **`value`** state is updated to store the next output value.

The next example multiplies its input by a constant:

```
defstar {
     name { FloatGain }
     domain { SDF }
     desc { Amplifier: output is input times "gain" (default 1.0). }
     input {
          name { input }
          type { float }
     }
     output {
          name { output }
          type { float }
     }
     defstate {
          name { gain }
          type { float }
          default { "1.0" }
          desc { Gain of the star. }
     }
     go {
          output%0 << double(gain) * double(input%0);
     }
}
```

The following example illustrates multiple inputs, ANYTYPE inputs, and the use of the **`print()`** method of the **`Particle`** class.

```
defstar {
     name { Printer }
     domain { SDF }
     desc {
Prints out one sample from each input port per line
If "fileName" is not equal to "cout" (the default), it
specifies the filename to write to.
     }
     explanation {
This star prints its input, which may be any supported type.
There may be multiple inputs: all inputs are printed together on
the same line, separated by tabs.
     }
     inmulti {
          name { input }
          type { ANYTYPE }
     }
     defstate {
          name { fileName }
          type { string }
          default { "cout" }
          desc { Filename for output. }
     }
     protected {
          UserOutput output;
     }
     setup {
          output.fileName(fileName);
     }

     go {
          MPHIter nexti(input);
          PortHole* p;
          while ((p = nexti++) != 0)
               output << ((*p)%0).print() << " ";
          output << "0;
     }
}
```

This star is *polymorphic* since it can operate on any type of input. Note that the default value of the output filename is ‘‘cout’’, which causes the output to go to the standard output.

4.7. Adding stars permanently to Ptolemy

If you create stars that are to be used by everyone in a group, it might be easiest if you make the star a permanent part of **Ptolemy**. Here is the procedure (assuming, of course, that you have permission to modify the **Ptolemy** official directories):

After you create your own star, you can install it into the appropriate star library in the following procedure:

1. Make the star icon (refer to the Pigi manual.)
2. Go to ˜ptolemy/src/domains/XXX/stars, and edit XXXstars.mk and make.template, adding a line for the star (XXX is the domain name).
3. Rebuild the system (change to ˜ptolemy and type "make").

Sometimes, you may want to modify an existing star. Besides modifying the code itself, you should reflect your changes in the documents which contain that star description and in the demo programs. First of all, using Pigi, correct the demo programs which contain the star. To update the documentation, you may want to look at:

(1) XXX{star_name}.pl, **explanation** item.

(2) domains/XXX/doc/demos. Some demos may use the modified star, and have the out-of-date description.

(3) domains/XXX/doc/overall. The domain manual may use the modified star as an example.

Where possible, it is preferable to ‘‘modify’’ a star by deriving another star from it that has the increased functionality than by replacing the existing star.

5. Appendix: Messages in Ptolemy

Authors:

Joseph Buck

5.1. Design Criteria for the Packet Interface

The Ptolemy heterogeneous message interface is designed to provide a mechanism that allows stars to transmit message that are arbitrary objects to other stars. The design was constrained by the following criteria:

- Existing stars (stars that were written before the message interface was added) that handle ANYTYPE work with message particles without change.
- Message portholes should be able to send different types of messages during the same simulation (just as a real network does).
- The design should be efficient: we should not repeatedly be required to copy large messages. This suggests using a reference-count mechanism as is used in many C++ classes (for example, string classes).
- At the same time, it should be possible to safely modify large messages without excessive memory allocation and deallocation.
- It should be easy for users to define their own message types, and no change to the kernel should be required to transmit the new message types.

To accomplish this goal, three classes were designed:

- The **`Message`** class is the base class from which all other message datatypes are derived. A user who wishes to define an application-specific message type creates a new class that is derived from **`Message`**.
- The **`Envelope`** class is an envelope; it "holds" an object derived from class **`Message`**. **`Envelope`** objects may be copied or duplicated; when this happens, the contents are not copied (several **`Envelope`** objects share the same **`Message`** object); at any time, the **`Message`** object's reference count says how many **`Envelope`** objects refer to it. When the last reference is removed, the contents are deleted.
- The **`MessageParticle`** class is a type of **`Particle`** (like **`IntSample`**, **`FloatSample`**, etc); it contains a **`Envelope`**. Most users will never see this class directly; if a **`PortHole`** is declared to be of type ‘‘message’’ it will use this type of **`Particle`**.

Class **`Particle`** contains two member functions for message support: **`getMessage`**, to receive a message, and the "<<" operator with a **`Envelope`** as the right argument, to load a message into a particle. These functions return errors in the base class; they are overridden in the **`MessageParticle`** class with functions that "do the right thing".

This interface was formerly called the packet interfaces; the classes have been renamed.

5.2. Defining A New Message Class

The designer of a new type of message does so by defining a new class that is derived from class `Message`. Certain virtual functions defined in that class must be overriden; others may optionally be overriden. Here is an example of a user-defined message type:

```
// This is a simple vector message body.  It stores
// an array of integer values of arbitrary length.
// The length is specified by the constructor.

#include "Message.h"

class IntVecData : public Message {
private:
     int len;

     init(int length,int *srcData) {
          len = length;
          data = new int[len];
          for (int i = 0; i < len; i++)
               data[i] = *srcData++;
     }

public:
     // the pointer is public for simplicity
     int *data;

     int length() const { return len;}

     // functions for type-checking
     const char* dataType() const { return "IntVecData";}

     // isA responds TRUE if given the name of the class or
     // of any baseclass.
     int isA(const char* typ) const {
          if (strcmp(typ,"IntVecData") == 0) return TRUE;
          else return Message::isA(typ);
     }

     // constructor: makes an uninitialized array
     IntVecData(int length) : len(length) {
          data = new int[length];
     }

     // constructor: makes an initialized array from a int array
     IntVecData(int length,int *srcData) { init(length,srcData);}

     // copy constructor
     IntVecData(const IntVecData& src) { init(src.len,src.data);}

     // clone: make a duplicate object
     Message* clone() const { return new IntVecData(*this);}
```

```
          // destructor
          ~IntVecData() {
                delete data;
          }
     };
```

This message body can contain a vector of integers of arbitrary length. Some functions in the class are arbitrary and the user may define them in whatever way is most convenient; however, there are some requirements.

The class must redefine class **`Message`**'s **`dataType()`** function. This function returns a string identifying the message type. This string should be identical to the name of the class. In addition, the **`isA()`** function must be defined. The **`isA`** Function responds with TRUE (1) if given the name of the class or of any baseclass; it returns FALSE (0) otherwise. This mechanism permits stars to handle any of a whole group of message types, even for classes that are defined after the star is written.

Because of the regular structure of **`isA`** function bodies, macros are provided to generate them. The **`ISA_INLINE`** macro expands to an inline definition of the function; for example,

```
          ISA_INLINE(IntVecData,Message)
```

could have been written above instead of the definition of **`isA`** to generate exactly the same code. Alternatively, to put the function body in a .cc file, one can write

```
               int isA(const char*) const;
```

in the class body and put

```
     ISA_FUNC(IntVecData,Message)
```

in the .cc file.

The class must define a copy constructor, unless the default copy constructor generated by the compiler, which does memberwise copying, will do the job.

The class must redefine class **`Message`**'s **`clone()`** function. Given that the copy constructor is defined, the form shown in the example, where a new object is created with the **`new`** operator and the copy constructor, will suffice.

In addition, the user may optionally define type conversion and printing functions if they make sense. If a star that produces messages is connected to a star that expects integers (or floating values, or complex values), the appropriate type conversion function is called. The base class, **`Message`**, defines the virtual conversion functions **`asInt()`**, **`asFloat()`**, and **`asComplex()`** and the printing function **`print()`** – see the Message.h file for their exact types. The base class conversion functions assert a run-time error, and the default print function returns a **`StringList`** saying

<*type*>: no print method

where *type* is whatever is returned by **`dataType()`**.

By redefining these functions, you can make it legal to connect a star that generates messages to a star that expects integer, floating, or complex particles, or you can connect to a Printer or Xgraph star (for Xgraph to work, you must define the **`asFloat`** function; for Printer to work, you must define the **`print`** function).

5.3. Use of the Envelope class

The **`Envelope`** class serves as an envelope to hold objects of class **`Message`** or derived classes. Once a user's Message object is placed into an Envelope object, the Envelope takes over responsibility for managing its memory: maintaining reference counts, deleting the Message when it is no longer needed.

The constructor (which takes as argument a reference to **`Message`**), copy constructor, and assignment operator for **`Envelope`** manipulate the reference counts of the **`Message`** object inside the **`Envelope`** objects. Assignment simply copies a pointer and increments the reference count. Whenever the destructor for a **`Envelope`** is called, the reference count of the contents is decremented; if it reaches zero, the contained **`Message`** object is deleted. Because of this deletion, a **`Message`** must never be put inside a **`Envelope`** unless it was created with the **`new`** operator. Once a **`Message`** object is put into a **`Envelope`** it should never be deleted; it will ‘‘live’’ as long as there is at least one **`Envelope`** that contains it and will then be deleted automatically.

It is possible for an **`Envelope`** to be ‘‘empty’’. If it is, the **`empty()`** member function will return true, and the data field will point to a special ‘‘dummy message’’ whose type is DUMMY and that has no data in it.

The **`dataType()`** member function of **`Envelope`** returns the datatype of the contained **`Message`** object; the functions **`asInt()`**, **`asFloat()`**, **`asComplex()`**, and **`print()`** are also ‘‘passed through’’ in a similar way to the contained object.

Two functions are provided for convenience to make type checking simpler: **`typeCheck`** and **`typeError`**. A simple example illustrates their use:

```
if (!envelope.typeCheck("IntVecData")) {
     Error::abortRun(*this,envelope.typeError("IntVecData"));
     return;
}
```

`typeCheck` calls **`isA`** on the message contents and returns the result, so an error will be reported if the message contents are not IntVecData and are not derived from IntVecData. Since the above code segment is so common in stars; a macro is included in Message.h to generate it; the macro

```
TYPE_CHECK(envelope,"IntVecData");
```

expands to essentially the same code as above and is shorter to type. **`typeError`** generates an appropriate error message:

Expected message type ’*arg*’, got ’*type*’

To access the data, two functions are provided: **`getData()`** and **`writableCopy()`**. The **`getData`** function returns a pointer to the contained **`Message`**-derived object. *The data pointed to by this pointer must not be modified,* since other **`Envelope`** objects in the program may also contain it. If you convert its type, always make sure that the converted type is a pointer to const (see the programming example for UnPackInt below). This will ensure that the compiler will complain if you do anything illegal.

The **`writableCopy`** function also returns a pointer to the contained object, but with a difference. If the object's reference count is one, the envelope is emptied (set to the dummy message) and the contents are returned. If the reference count is more than one, a *clone* of the contents is made (by calling its **`clone()`** function) and that is returned; again the envelope is zeroed (to avoid the making of additional clones later on).

In some cases, a star writer will need to keep a received **`Message`** object around between executions. One way to accomplish this is for the star to have a member of type **`Envelope`**, and to use this member object to hold the message data between executions. Messages should always be kept in envelopes so that the user does not have to worry about managing their memory.

5.4. Use of the MessageParticle class

If a porthole is of type ''message'', then its particles are of the class **`MessageParticle`**. A MessageParticle is simply a particle whose data field is an Envelope, which means that it can hold a Message in the same way that Envelope objects do.

Many of **`Particle`**'s functions are redefined by this class to cause a run-time error; for example, it is illegal to send an integer, floating, or complex number to the particle with the '<<' operator. The conversion operators (conversion to type int, float, or Complex) return errors by default, but can be made legal by redefining the **`asInt`**, **`asFloat`**, or **`asComplex`** member functions for a specific message type.

The principal operations on **`MessageParticle`** objects are '<<' with an argument of type **`Envelope`**, to load a message into the particle, and **`getMessage(Envelope&)`**, to transfer message contents from the particle into a user-supplied message. **`getMessage`** removes the message contents from the particle. For cases where the destructive behavior of **`getMessage`** cannot be tolerated, an alternate interface, **`accessMessage(Envelope&)`**, is provided. It does not remove the message contents from the particle. Promiscuous use of **`accessMessage`** in systems where large-sized messages may be present can cause the amount of virtual memory occupied to grow (though all message will be deleted eventually).

5.5. Use of Messages in Stars

Here are a couple of simple examples of stars that produce and consume messages. For more advanced samples, look in the Ptolemy distribution for stars that produce or consume messages.

The reason for this ''aggressive reclamation'' policy (both here and in other places) is to minimize the number of no-longer-needed messages in the system and to prevent unnecessary clones from being generated by **`writableCopy()`** by eliminating references to **`Message`** objects as soon as possible.

```
defstar {
     name { PackInt }
     domain { SDF }
     desc { Accept integer inputs and produce IntVecData messages.}
     defstate {
          name { length }
          type { int }
          default { 10 }
          desc { number of values per message }
     }
     input {
          name { input }
          type { int }
     }
     output {
          name { output }
          type { message }
     }
     ccinclude { "Message.h", "IntVecData.h" }
     start {
          input.setSDFParams(int(length),int(length-1));
     }
     go {
          int l = length;
          IntVecData * pd = new IntVecData(l);
          // Fill in message.  input%0 is newest, must reverse
          for (int i = 0; i < l; i++)
               pd->data[l-i-1] = int(input%i);
          Envelope pkt(*pd);
          output%0 << pkt;
     }
}
```

Since this is an SDF star, it must produce and consume a constant number of tokens on each step, so the message length must be fixed (though it is controllable with a state). Notice that the output porthole is declared to be of type message. Notice also the **`ccinclude`** statement; we must include the file Message.h in all message-manipulating stars, and we must also include the definition of the specific message type we wish to use.

The code itself is fairly straightforward -- an **`IntVecData`** object is created with **`new`**, it is filled in with data, put into a **`Envelope`** envelope, and sent. Resist the temptation to declare the **`IntVecData`** object as a local variable; it won't work.

And here's a star to do the inverse operation:

```
defstar {
     name { UnPackInt }
     domain { SDF }
     desc {
Accept IntVecData messages and produce integers.  The first ’length’
values from each message are produced.
     }
     defstate {
          name { length }
          type { int }
          default { 10 }
          desc { number of values output per message }
     }
     input {
          name { input }
          type { message }
     }
     output {
          name { output }
          type { int }
     }
     ccinclude { "Message.h", "IntVecData.h" }
     start {
          output.setSDFParams(int(length),int(length-1));
     }
     go {
          Envelope pkt;
          (input%0).getMessage(pkt);
          if (!pkt.typeCheck("IntVecData")) {
               Error::abortRun(*this,pkt.typeError("IntVecData"));
               return;
          }
          const IntVecData * pd = (const IntVecData *)pkt.myData();
          if (pd.length() < int(length)) {
               Error::abortRun(*this,"Received message is too short");
               return;
          }
          for (i = 0; i < int(length); i++) {
               output%(int(length)-i-1) << pd->data[i];
          }
     }
}
```

Because the domain is SDF, we must always produce the same number of outputs regardless of the size of the messages. The simple approach taken here is to require at least a certain amount of data or else die.

The operations here are to declare an envelope, get the data from the particle into the envelope with `getMessage`, check the type, and then access the contents. Notice the cast operation; this is needed because **`myData`** returns a const pointer to class **`Message`**. It is important that we converted the pointer to **`const IntVecData *`** and not **`IntVecData *`** because we have no right to modify the message through this pointer. Most C++ compilers will not warn by default about ‘‘casting away const’’; we recommend turning on compiler warnings when compiling code that uses messages to avoid getting into trouble (for *g++*, say –**`Wcast-qual`**; for *cfront*-derived compilers, say **`+w`**).

If we wished to modify the message and then send the result as an output, we would call **`writableCopy`** instead of **`myData`**, modify the object, then send it on its way as in the previous star.

6. Domains in Ptolemy

Authors:

Joseph Buck
Soonhoi Ha
Edward A. Lee
David G. Messerschmitt

6.1. Introduction

One of the key objectives of the **Ptolemy** design environment is to support the coexistence and interaction of diverse computational models, called *domains*. For instance, a dataflow domain can exist within a discrete-event domain, so that for example a signal processing subsystem can be cleanly incorporated within a communication network simulation. This mixing of domains is one of the primary research objectives of the **Ptolemy** project. We do not consider all problems solved, and some details of the domain construction and inter-domain interface may change in the future.

In the **Ptolemaic** cosmology, elementary functional blocks are called **`Star`**s, higher level blocks are called **`Galaxy`**s, and an entire application is called a **`Universe`**. The domain of a star is specified in the star's definition and determines its class ancestry. For example, all stars in the SDF domain are derived from the **`SDFStar`** class. The domain of a galaxy or universe must be explicitly specified by setting its domain property. The command in the **Ptolemy** graphical interface for doing this is ‘‘edit-domain’’ and in the interpreter is ‘‘domain *name*’’. If not otherwise specified, the domain of a galaxy is the same as that of the galaxy or universe it sits within. If the domain is specified to be different than that of the galaxy or universe it sits within, then the galaxy is realized as an object called a **`Wormhole`**. Once a **`Wormhole`** is created, it behaves externally like a **`Star`**. In other words, to the external domain, it looks like a monolithic object with no internal structure of interest. Internally, however, it encapsulates an entire foreign domain invisible from the outside domain. By this mechanism, the internal computational model can be totally different from the external model, in that the specification language, operational semantics, and scheduling paradigm can be different. Wormholes, therefore, are the primary mechanism in **Ptolemy** for supporting heterogeneity.

When the scheduler of the outer domain fires the wormhole , the inner scheduler is invoked for the inner galaxy until some specified stopping condition is met. At that time, control returns to the scheduler of the outer domain. Of course, a wormhole may contain other wormholes. This document explains the mechanism of the domain interface, and documents how to create new domains in **Ptolemy**. Of necessity, the explanation is somewhat sketchy. To write a new domain, using the source code of an existing domain as a model is a good idea.

6.2. Domain Design

Data abstraction and polymorphism, two tenets of object-oriented programming, allow domains (corresponding to models of computation) to be abstracted so that their differences are not visible from the kernel. Domains can be selectively linked into **Ptolemy**, and the mere action of linking them makes them visible and usable. **Ptolemy** executables can therefore be created that are tuned to a particular application. For example, if we are interested only in simulation of communication networks, we can link only the DE and the SDF domains to the kernel, excluding other domains.

To define a new domain, it is necessary to define a base class for the stars in the domain, a set of classes for interconnecting stars in the domain and passing data between them, a scheduler, one or more targets, a wormhole, a mechanism for connecting to other domains through the wormhole boundary, and a domain object. This is no small task. However, all of these objects have well-defined base classes from which the new classes must be derived, and most new domains make only very minor additions or changes to these base classes. Major changes, however, are possible, and may be required to introduce domains that operate in very different ways compared to the existing domains.

To make the task of producing a new domain a bit easier, we include a sample ‘‘dummy domain’’ named ‘‘XXX’’ in the directory ‘‘˜ptolemy/src/domains/xxx/kernel’’.

There is a great deal of flexibility in the definition of new domains, and only the **Ptolemy** source code can provide full documentation. If the *ptlang* preprocessor is to be used with a new domain, however, then there is less flexibility. This preprocessor generates C++ code that assumes that certain classes have been defined. These classes normally contain the domain name as part of their class name, e.g. **`SDFStar`**, and **`InSDFPort`**. Hence when defining a new domain, many new classes have to be defined. Of course, if these are identical to existing classes, the definition may be very simple, as in the following example:

```
#define XXXGeodesic AutoForkNode
```

In this example, a **`Geodesic`** has been defined for a new domain ‘‘XXX’’ to be identical to a class defined in the kernel, **`AutoForkNode`**.

A domain has a specific operational semantics for a block diagram representation. The operational semantics are realized by a **`Scheduler`** for the domain, so in some sense this forms the heart of the domain. The scheduler determines when to fire the blocks in an application. This determination can be made statically or dynamically. For that purpose, the **`Scheduler`** base class has two methods: *setup()* (for operations done once before a run, such as static scheduling), and *run()* (for dynamically invoking blocks). The *run()* method fires blocks under its control until a stopping condition is met. Two different methods are provided to specify the stopping condition for the run. If the domain is that of a **`Universe`**, we call the *setStopTime()* method. If the domain resides in a **`Wormhole`**, we call the *resetStopTime()* method.

Control of execution of a universe is accomplished by a **`Target`**. In most simulation domains, the **`Target`** object simply passes commands through to the **`Scheduler`** object; it may also be used to pass parameters through to the scheduler. In code generation domains, in particular, different **`Target`** objects model different targets for the generated code.

It may also be necessary to define domain specific star and **`PortHole`** classes. For example, the stars and portholes in the DE domain need to keep timing information while those in the SDF domain need not. They may also have some members and methods accessed by the **`Scheduler`** in the domain. An example is the *repetitions* in the **`SDFStar`** class, which indicates how many times the star will be executed in one iteration. It is required by the scheduler during the setup phase to perform static scheduling. Similarly the **`SDFPortHole`** class contains methods for setting parameters that are specific to SDF, such as the number of **`Particle`**s consumed or produced on a porthole when the star fires.

To get a new domain to interact with existing domains, it is necessary to define **`Wormhole`** and **`EventHorizon`** classes. It is worth noting that a domain may do without the **`Wormhole`** class if the domain will never contain other domains inside. However, the **`EventHorizon`** class is required for the domain to be an inner domain of a wormhole. An event horizon actually consists of a pair of classes, **`XXXtoUniversal`** and **`XXXfromUniversal`**, where **`XXX`** is the domain name. These classes are placed in the input or output port of a wormhole depending on the direction of dataflow. Further details of these classes are given below.

An additional class defined in a domain is the **`Domain`** class. Its definition are straightforward but essential for domain abstraction in the kernel. The source code for the base class in the kernel is self-explanatory, so it serves as the best reference. Wormhole design, however, is much more subtle, and hence is explained in more detail below.

A complete domain definition also consists of a library of stars. However, the design of these stars is entirely up the domain designer, so there is little guidance we can provide. Note that for many new domains, it will be possible to use the *ptlang* preprocessor unchanged for the design of stars. This is encouraged because it promotes uniformity. The preprocessor also produces documentation for stars.

6.3. Timing Relationship Between Domains

Ptolemy can support an effectively unlimited number of different domains since adding a new domain affects neither any existing implementation of other domains nor the kernel. The domain interface should take care of timing relations between the outer and the inner domains. Timing management is the most challenging task in the design of **`Wormhole`**s and their interconnection mechanism. Our approach is to classify the domains into two groups: *timed*, and *untimed*. In a *timed* domain, the scheduler keeps track of the global timing relations among data packets **`(Particles).`** On the other hand, an *untimed* domain requires only the local ordering information of data packets. The taxonomy of existing domains is:

timed : DE, THOR, etc
untimed : SDF, DDF, CG, etc

The scheduler of a domain maintains a member called *currentTime* for timing management. For a timed domain, *currentTime* indicates the global time. As the scheduler proceeds, *currentTime* also increases. Hence, the stopping condition for a timed domain is given by a stopping time. On the other hand, an untimed domain does not have the concept of time in its model of computation. If an untimed domain resides in a **`Wormhole`**, then the *currentTime* of the domain is simply the copy of the *currentTime* of the outer domain.

There are four combinations of timed and untimed domains. For each combination, we will show how to do timing management.

6.3.1. An Untimed Domain in an Untimed Domain

This first combination is the simplest for timing management: an untimed domain contains another untimed domain. For example, a DDF domain may contain an SDF domain. Since the outer domain does not care about the *currentTime* of the inner domain, we are free to manage the *currentTime* of the inner domain. In this case, the *currentTime* of the inner domain is just copied from that of the outer domain. During the execution of the wormhole, the *currentTime* remains unchanged. The stopping condition of the inner domain is given by the number of iterations of the schedule. The *resetStopTime()* method of the scheduler sets the stopping condition to "one iteration", ignoring the argument to the method. The meaning of "one iteration" is domain specific. Refer to the individual domain documentation for the interpretation of "one iteration".

6.3.2. An Untimed Domain in a Timed Domain

The second possibility is that a timed domain contains an untimed domain. If a DE domain contains an SDF domain, it falls into this category. In this case, the *currentTime* of the inner untimed domain is synchronized with the *currentTime* of the outer timed domain at the beginning of the execution of the wormhole. Like the first case, the *currentTime* of the inner domain is not increased during the execution. It gives an illusion to the outer domain that the untimed domain takes zero time for execution. Since the inner domain has no notion of time, it seems not only awkward but infeasible to assign any execution delay to the inner domain. Instead, we can simulate any desired time delay of the inner domain by attaching a **`Delay`** star in the outer domain to the inputs or outputs of the wormhole. The decoupling of the delay and the functionality of a block in a timed domain is the basic technique we use in a timed domain like DE. The stopping condition of the inner domain is again "one iteration".

6.3.3. A Timed Domain in a Timed Domain

This third combination is the most complicated: a timed domain contains another timed domain. It is complicated because timed domains have a notion of chronology, but the principle of wormholes implies that the chronology of one domain cannot be visible from another. Maintaining consistency between these chronologies is the principal challenge. A DE domain containing a THOR domain will be an example. First of all, since both domains have global clocks, we have to synchronize them. Therefore, the inner domain’s **`Target`** object has a **`State`** called *timeScale* which is the ratio between the time unit of inner domain and that of the outer domain. The *timeScale* is 1.0 by default (no scaling). At the beginning of the execution of the wormhole, the *currentTime* of the outer domain is scaled by the *timeScale* and copied to the *currentTime* of the inner domain. The *currentTime* of the inner domain proceeds as the wormhole is executed until the stopping condition is met. After the execution finishes, the *currentTime* of the inner domain is scaled again by the inverse of the *timeScale* and informed to the outer domain. A user can change the value of the scale factor by changing the value of the target parameter (*timeScale*) in the outermost galaxy of the inner timed domain. In the interpreter, the following line does it:

```
targetparam timeScale 10.0
```

In the graphical interface, this state can be modified by executing the *edit-target* command on the inner galaxy. Doing so causes two menus to appear in succession; the first permits the user to change the target object used; simply click ‘‘OK’’. The second menu permits galaxy parameters to be altered; now the *timeScale* parameter can be altered.

The decision about the stopping condition of the inner domain is the most complex part of the domain interface. Suppose that the wormhole has two inputs ($input_1$, $input_2$) and two outputs ($output_1$, $output_2$). If $input_1$ receives an event **`(Particle),`** the wormhole starts execution. The complication arises from the observation that this wormhole cannot safely process internal events with later time stamps than the event on $input_1$. It may later receive an event on $input_2$ that invalidates these computations. For example, if $input_2$ is a reset input of a system, the reset event will affect the previous execution if it is not completed. Therefore, we can process the inner system at a given event only until the time stamp of the next event to arrive at any input. Unfortunately it is generally not possible to predict when the next event will arrive. One possible solution is to process the wormhole for one time unit at each invocation. In case of THOR domain, one time unit is well defined so that this approach is acceptable. However, there is no notion of a basic time unit in the DE domain. Also, the approach is too conservative and will result in a lot of overhead. Even though there will be no event during the next 100 time units, the wormhole would be invoked 100 times to march one time unit at each invocation.

Another approach is to process the wormhole until the next earliest event in the outer domain. When the wormhole is to be executed, we check the *expected* next global time by examining the next earliest event in the event queue of the outer domain. The stopping condition of the inner timed domain is set with the stop time equal to the expected next global time. As long as the wormhole does not generate any event during execution, the expected next global time becomes the *actual* next global time. If the inner domain generates any output events, however, they may be the next earliest events in the outer domain. Hence, this approach may not work. Suppose that the inner domain generates output events through $output_1$ earlier than through $output_2$. Moreover, the output event through $output_1$ is routed back to $input_2$ of the **`Wormhole`**. Then, we meet the original problem if this feedback event should have affected the output event through $output_2$ of the previous execution. Since the output event through $output_2$ is already produced in the previous execution, there is no way to correct the event.

We can improve the previous approach as follows. Suppose that we have exactly one timed **`Wormhole`**. At the beginning of the execution, we set the stop time of the **`Wormhole`** with the expected next global time. We proceed the inner scheduling until the *currentTime* reaches the stop time, or until the inner domain generates any output. If the wormhole generates an output, the output becomes the next earliest event in the outer domain. Therefore, our scheme is simply to process the wormhole until the actual (not expected) next global time in the outer domain. This proves the correctness of our solution. But, if there are more than one, say two, timed wormholes, this approach also fails. We have to compare the time stamps of the output events from both wormholes to detect the actual next global time. While we are executing a wormhole, it is not possible to determine the actual next global time since it is unknown before executing the other wormhole.

Finally, we have developed the following solution. When a wormhole is fired, its stopping time is set equal to the external global time. Generally that external global time must be equal to the time stamp of the event that triggered the firing of the wormhole. Thus, the stopping time is equal to the time stamp of the event that triggered the firing of the wormhole. Thus, generally speaking the wormhole cannot advance in time at all. The wormhole looks like a functional star without delay. The wormhole processes this single input event and any simultaneous internal events, sets its global time to the oldest (stalest) event on its event queue, and returns control. The external domain must invoke this wormhole again at this internal global time, or earlier if another event on its own event queue triggers the wormhole earlier. Thus, in effect the external domain must store an event on its event queue at the internal global time for the wormhole; in effect, the outer event queue is storing the stalest event on the inner domain event queue. This event triggers the firing of the wormhole at that proper time. Therefore, at the end of the execution of the wormhole, we determine the next event time of the inner domain and copy it to *currentTime* of the inner scheduler. We also set a flag, *stopBeforeDeadlocked*, if the inner domain is not deadlocked. If the flag is set, Then, the outer scheduler stores a triggering event of the inner domain in the event queue at the scaled *currentTime* of the inner scheduler.

In this approach, the total system event queue consists of the outer domain event queue and the wormhole event queues. These event queues are coordinated by storing the stalest event of the inner domains on the outer domain event queue in order to invoke the wormholes at the proper time. Each time a wormhole is fired, it processes one event, either an external triggering event or an internal event queue event. If there are multiple events with the same time stamp, they can all be processed.

An important exception occurs when the outer domain knows that future events are constrained in the earliest time they can occur. For this case, the outer domain can set a stopping time equal to that "earliest next event". This will be more efficient because generally the wormhole can process more than just the external triggering event, it can also process some internally generated and consumed events. This exception would occur for an SDF outer domain, which when it invokes the wormhole can set the stopping time equal to the next sample time for the wormhole.

In summary, the inner domain can never overtake the the outer domain in time, which proves the correctness of the approach regardless of the number of timed wormholes involved. The timed wormhole forms a special class of stars in a timed domain, called "process-star". The first invocation of a process-star is triggered by external events to the star. But the further invocations are triggered without external events until the star has no more internal events to process.

6.3.4. A Timed Domain in an Untimed Domain

The last possible combination is that of a timed domain in an untimed domain. Even though the untimed domain has no notion of time, it needs to control the inner timed domain by setting the stopping condition. For that purpose, an untimed domain keeps a property called its *schedulePeriod*. The *currentTime* of the untimed domain is increased by the *schedulePeriod* after each iteration. A user can control the value of the *schedulePeriod* by altering the (*schedulePeriod*) target parameter in the outermost galaxy of the untimed domain. In the interpreter, the following line does the job:

```
targetparam schedulePeriod "10.0"
```

The default value of *schedulePeriod* of the untimed domain is set to a large number (10000). Presumably, the *schedulePeriod* has no physical significance since in the untimed domain, time has no meaning. The sole purpose of the state is to define the stopping condition of the timed domain. A user should make sure that the wormhole does all necessary tasks before it reaches the stopping condition. The common approach is to set the stop time large enough so that the timed domain is deadlocked (meaning that there are no active events in the timed system) before the stop time. Then, the timed domain acts as a functional block like a conventional star in the untimed domain. Note that in this case, the inner timed domain need not stop after it generates its first output event, unlike the third case above.

There is a special case where the above discussion is not applicable. Suppose that the untimed universe is an SDF domain. We can interpret the SDF domain as a time driven domain where the time interval between data packets is fixed, and hidden safely in the system representation. A synchronous digital signal processing system falls into this category. In this case, the SDF domain looks like a timed domain!. The *schedulePeriod* then has physical significance. Hence, the user has to supply a meaningful value. The *schedulePeriod* of the SDF system represents the repetition period of the SDF schedule, and during one iteration, there may arrive more than one data packet at the wormhole. The time-interval of the data packets becomes the ratio of the *schedulePeriod* and the number of arrived data packets during one iteration. It is this time-interval and not the *schedulePeriod* that determines the stop time of the inner timed domain. This combination of a SDF domain and a timed domain has abundant applications.

6.3.5. Summary

We have discussed four combinations of the domains. If a wormhole contains another wormhole, we can apply the basic rule developed in this section to each domain interface. For example, a timed domain contains a wormhole of an untimed domain which itself contains a timed wormhole. Since the timed domain inside an untimed domain loses the physical meaning of times, the innermost timed domain has nothing to do with the outer timed domain in timing management. Still the untimed domain takes zero time in the sense of the outer global clock. An exceptional case is when the untimed domain is the SDF domain for time-driven application (e.g. DSP application). By controlling the *schedulePeriod* of the SDF domain, we can synchronize the innermost timed domain and the outer timed domain.

6.4. The Wormhole Class

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`**. The **`Target`**, in turn, has a pointer to a **`Scheduler`**. 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.

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

`Scheduler`.

The **`Scheduler`** executes a star by calling its **`run()`** method. This method first causes data packets **(`Particles`**) to be delivered to the **`Star`** by invoking the **`receiveData()`** method of the input ports. The **`go()`** method of the star is then invoked, which does whatever computation is associated with this particular star. After the execution of **`go()`**, the **`run()`** method then sends particles to other stars by calling the **`sendData()`** method of the output ports. Thus, the **`run()`** method calls a sequence of methods to execute a star: **`receiveData()`** on the input ports, **`go()`** on the **`Star`**, and **`sendData()`** on the output ports. The same sequence of calls are also used when the star happens to be a wormhole. In this case **`receiveData()`** and **`sendData()`** operate via the event horizon. The design of these methods will be discussed in the next section.

The main part of the domain interface is performed in the

`go()` method of the **`Wormhole`**. As an example, the **`go()`** method of the **`SDFWormhole`** class is defined as follows.

```
void SDFWormhole :: go() {
      // set the currentTime of the inner domain.
      target->setCurrentTime(arrivalTime);
      // run
      Wormhole::run();
}
```

After getting the input data packets (**`receiveData()`**), the **`Wormhole`** first synchronizes the inner domain with the outer domain. The **`currentTime`** of the inner target and scheduler is set to the arrival time of the data packets **(`arrivalTime)`**. Since the SDF domain is untimed, the decision on the arrival time of the data packets is trivial. The next step is to call the **`run()`** method of the base **`Wormhole`** class. Here is a part of the definition[7] of the **`Wormhole`** class.

---

[7] We omit the members that are not relevant to the domain interface such as the constructor and print methods.

```
class Wormhole : public Runnable {
public:
     void setup() { initTarget(); }
     int run() {
          if (!checkReady()) return TRUE;
          setStopTime(1.0);    // dummy value
          if (!Runnable :: run()) return FALSE;
          sumUp();
          return TRUE;
     }

     // redefine setStopTime()
     void setStopTime(double) { target->resetStopTime(getStopTime()); }
protected :
     // get the stopping condition for the inner domain.
     // Must be redefined in the derived class.
     virtual double getStopTime() = 0;

     // arrange things after run if necessary
     virtual void sumUp() {}

private :
     // check ready
     int checkReady() const;
};
```

The **`run()`** method of the **`Wormhole`** class consists of four method calls. The first one, **`checkReady()`**, checks the readiness of the input data. Suppose the wormhole is a DE wormhole, and the inner domain is an SDF domain. If there is an input event to the wormhole, then the wormhole is runnable[8] from the point of view of the DE domain, and the **`go()`** method is called. However, the inner SDF domain may not be fired before all input **`PortHole`**s have enough particles. In this case, the **`checkReady()`** method will return FALSE to terminate the execution. This method is discussed in more detail below.

When all input data packets are ready, we can initiate the inner scheduler by calling the **`run()`** method of the **`Runnable`** class which is the base class of the **`Wormhole`**. Before initiating the inner scheduler, we should specify its stopping condition using the **`setStopTime()`** method of the wormhole, which invokes the **`resetStopTime()`** method of the scheduler. How to set the value of the stop time is discussed above.

In the initialization stage of the outer galaxy, the **`setup()`** method is called. The **`setup()`** method of a wormhole calls the **`setup()`** method of the base **`Wormhole`** class (see for instance the source code for **`SDFWormhole`**:: in ˜ptolemy/src/domains/sdf/kernel). In the **`setup()`** method, the inner domain is not only initialized but also any compile-time scheduling for the inner domain is performed. Also, the flag **`stopBeforeDeadlocked`** is

---

[8] The decision of runnability of a wormhole is indistinguishable from those of other stars. For example, an **`SDFWormhole`** is runnable when all input ports have sufficient data packets.

set FALSE by default. The flag will be set TRUE by the inner scheduler when the inner domain suspends execution before being deadlocked (meaning that one or more stars in the domain is runnable).

6.5. EventHorizons

Like a **`Star`**, a **`Wormhole`** may have input and output **`PortHole`**s. The portholes of the wormhole are called *event horizons*, named after the point at which an object that falls into a black hole disappears from our universe (and presumably reappears in another). These objects are defined using multiple inheritance: for each such object, one baseclass is some type of **`PortHole`** and the other baseclass is either **`ToEventHorizon`** or **`FromEventHorizon`**, each of which are derived from the class **`EventHorizon`**.

Precisely speaking, one porthole of a wormhole consists of a pair of event horizons, one corresponding to the outer domain, and the other corresponding to the inner domain. For example, suppose an SDF application has a wormhole **`(SDFWormhole)`** which contains a DE subsystem. Then, an input port of the wormhole comprises a **`SDFtoUniversal`** event horizon and a **`DEfromUniversal`** event horizon. The former is connected to the outer domain (to some **`SDFStar`**) and the latter is connected to the inner domain (to some **`DEStar`**). Similarly, an output port of the wormhole comprises a **`DEtoUniversal`** event horizon for the inner connection and a **`SDFfromUniversal`** event horizon for the outer connection.

The **`EventHorizon`** class has two derived classes: **`FromEventHorizon`** and **`ToEventHorizon`**. In general, a domain has a pair of derived event horizons, one derived from the **`FromEventHorizon`** class and the **`XXXPortHole`** class, where **`XXX`** is the domain (e.g. **`SDFfromUniversal, DEfromUniversal`**), and the other from the **`ToEventHorizon`** class and the **`XXXPortHole`** class (e.g. **`SDFtoUniversal, DEtoUniversal`**). The **`receiveData()`** method of the **`ToEventHorizon`** class receives data packets from its domain and transfers them to the counterpart **`FromEventHorizon`** class. Similarly, the **`sendData()`** method of the **`FromEventHorizon`** class receives data packets from the counterpart **`ToEventHorizon`** and transfers them to its domain. Therefore, the data transfer between domains is performed by calling the **`receiveData()`** of the **`ToEventHorizon`** first, and calling the **`sendData()`** of the **`FromEventHorizon`** next, regardless of whether it happens at an input port or an output port of the wormhole.

If data transfer were the sole purpose of the event horizon, we would only need one **`EventHorizon`** class. Unfortunately, there are some domain-specific operations we need to do in the event horizon. There are two alternatives to perform the domain-specific operations. First, We may optimize each domain interface by defining a **`EventHorizon`** class for each pair of domains. This approach is not attractive for the highly modularized and easily expandable design of **Ptolemy**. If we have N domains, we would need to define N^2 **`EventHorizon`**s. To make matters worse, if a user wants to add a new domain into **Ptolemy**, he or she would need to know the details of the existing domains to design the **`EventHorizon`**s, and those designs might need to be modified. Hence, we have implemented the approach of a ‘‘universal event horizon’’ where each domain has only a pair of event horizons. Now, we have only $2N$ **`EventHorizon`**s for N domains, and each domain is totally independent of each other. This mechanism depends on the universality of the universal event horizon; we consider this approach still experimental.

While data transfer between two domains is the basic function of event horizons, they also need to manage time stamps for domains that have them. Some domains in **Ptolemy** associate a time stamp with each particle and other domains not. Since a domain does not know with which domain it is interacted, the **`EventHorizon`** class of the domain must use a time stamp regardless of the domain. Therefore, the time stamp of a **`ToEventHorizon`** is set to be the arrival time of the data packet from its domain, and the time stamp of a **`FromEventHorizon`** is set equal to that of the paired **`ToEventHorizon`**.

A pair of event horizons for a domain may have other domain-dependent responsibilities. One example can be found in **`DEfromUniversal`** class. This class not only receives particles but also puts events into the global event queue in the DE domain (refer to the **DE Domain** chapter in the **Almagest**). Another example is the **`ready()`** method in the **`FromEventHorizon`** class. This method is to check whether the port is ready for the inner domain to be fired. For the SDF domain, it returns true only when it has sufficient data. Supposing that the inner domain is SDF and the outer domain is DE, the inner domain may not be executed until all input ports of the **`DEWormhole`** have enough data packets. For that purpose, the **`checkReady()`** method of the wormhole checks all **`FromEventHorizon`**s (using the **`ready()`** method) of the input ports whether they are ready. For other domains, the **`ready()`** method can be redefined for more elaborate synchronization, or ignored if no synchronization is necessary, as in the DE domain.

6.6. Status

Nine domains are included in the 0.4 release of **Ptolemy** : two timed domains (DE and THOR), and seven untimed domains (SDF, DDF, CG, CG-DDF, CGC, CG56, and CG96). We have developed other experimental domains that are not part of the distribution. The source code, demos, and documentation for these domains is contained in ˜ptolemy/src/domains/xxx, where ‘‘xxx’’ is the domain name (in lower case letters). These domains can be mixed arbitrarily to compose a complicated system. Any wormhole may contain any other wormhole to make a hierarchical structure (exception: it is not currently possible to define a wormhole whose outer domain is one of the code generation domains (CGxx) and whose inner domain is something else).

7. Ptolemy Documentation

Authors:

Edward A. Lee
Joseph T. Buck

7.1. Introduction

Since **Ptolemy** is by design an extensible system, the documentation must be also be extensible. To ensure reasonable portability, only standard tools in the Berkeley Unix[9] environment have been used. Furthermore, all shell scripts and Makefiles are supplied along with the documentation so that they can be modified if necessary. The formating is with *ditroff* (device independent troff) with the -me macros, *eqn*, and *tbl*. In addition, some custom macros have been supplied in order to enforce a consistent look across all **Ptolemy** documents. These macros and their use are explained here. The -me macros will not be covered, since they are adequately documented elsewhere.

Since not all Unix systems have *ditroff*, we also provide scripts for printing and previewing the documentation using *groff*, the freely redistributable replacement for *troff* from the Free Software Foundation. If you wish to use *groff*, follow the installation instructions at the end of this section.

7.2. Printing the manual

First, all scripts for printing the manual use the c shell, and require that the printer you wish to use be defined. To do this, place the following line in the file .cshrc in your home directory:

```
setenv PRINTER lw
```

where `lw` is replaced with whatever name your printer has.

All kernel documentation and utilities for printing documentation are contained in the directory ‘‘˜ptolemy/doc’’. It is divided into sections as follows:

main:	This directory documents the **Ptolemy** kernel and user interfaces.
bin:	The scripts for printing the manual are here.
headers:	The troff header files are here.

Domains are documented where their source code resides. For instance, the documentation for the SDF domain is in ‘‘˜ptolemy/src/domains/sdf/doc’’. Each directory with documentation has a Makefile. Printing subsections of the manual can be accomplished by typing "make print" in the appropriate directory. For instance, to print the whole manual,

---

[9] Unix is a trademark of AT&T.

```
% cd ˜ptolemy/doc
% make print
```

To print only section 1 of the manual, which contains this section,

```
% cd ˜ptolemy/doc/main
% make print
```

You can preview the document on the screen. For instance,

```
% cd ˜ptolemy/doc/main
% make preview
```

You can pass options to *ditroff*. For instance, suppose you wish to print only pages 48 through the end of section 1. Then use

```
make OPTIONS=-o48- print
```

Any set of *ditroff* options may be passed this way. To print only the pigi document,

```
% cd ˜ptolemy/doc/main/pigi
% make print
```

A table of contents for each major section will also be printed. Sorry, but this may not be printed in the proper order, so some rearranging of the pages may be required. If you are printing the entire manual, an index will be generated after the entire manual has been printed. If you have printed a new version of a subsection, and wish to update the index, then:

```
% cd ˜ptolemy/doc
% make prindex
```

When only a subsection is printed, every attempt is made to format it as a standalone document. For this reason, sections may be numbered differently than they would be in the complete manual. In addition, a title page may appear that does not appear when the complete manual is printed.

7.3. Writing new sections

In order to get consistent documentation, a set of troff macros is used to control most important formatting options. These macros are explained here. The set of macros used to document basic capabilities is different from the set used to document demos, stars, and targets. For stars, the formatted documentation will be generated automatically by the preprocessor. For demos and targets, you must generate it yourself.

7.3.1. Basic documents

An example of a basic document is this one, covering **Ptolemy** documentation. The source for this document is in ‘‘˜ptolemy/doc/main/documents’’. The macros it uses are defined in ‘‘˜ptolemy/doc/headers’’. In that directory, it uses macros defined in the file called ‘‘shared’’, and in either the file called ‘‘standalone’’ or ‘‘together’’, depending on whether the document is printed by itself or in combination with other manual sections.

A typical document has the following syntax:

```
.\" %W%   %G%
.VR 0.%I% %G%
.TI "SECTION TITLE"
.AU
Author list
one name per line
no punctuation
.AE
.H1 "Section Title"
.H2 "Title"
.H3 "Title"
.H4 "Title"
.UH "Un-numbered header"
```

Text appearing between headers uses standard -me macro formatting. The first line is a comment that will be translated by sccs into a version number for the document, looking something like:

```
.\" @(#)filename 1.1     10/14/90
```

This is interpreted by *ditroff* as a comment, and hence has no affect on the output. The version number that will appear on the output is specified by the .VR macro. It takes two arguments, a version number and a date. These two arguments will be printed in the lower left and right corners of each page, respectively. If the version number is given as ‘‘%I%’’, it will be translated by sccs into the same version number that follows ‘‘filename’’ in the ‘‘@(#)’’ line above. For instance, this document is version ‘‘1.14’’ (see the lower left corner), with a prepended number corresponding to major releases of **Ptolemy**. If the date is given by ‘‘%G%’’, it will be translated by sccs into a date as shown in the lower right hand corner of this document.

The ‘‘.TI’’ macro gives the section title. When the document is printed standalone, this title will be put on a separate title page. Otherwise, it will be a level-one header. The rest of the macros should be self explanatory. A table of contents will be generated using these headers.

7.3.2. Adding to main

If you are adding a new document to the ‘‘main’’ part of the **Ptolemy** documentation, contained in ‘‘˜ptolemy/doc/main’’, then you must edit the Makefile in ‘‘˜ptolemy/main’’. Simply add your new directory name to the definitions of the variables **`DIRS`** and **`SECTIONS,`** using existing entries as a model.

Your new directory should contain a file called ‘‘overall’’ which defines the entire document. It may contain lines like

```
.so filename
```

to source from another file. Typically, all major subsections of the document would be written in their own file, with a corresponding ‘‘.so’’ entry in ‘‘overall’’.

Your new directory should also itself contain a Makefile, used for printing the document in standalone form. This should be a symbolic link to the file ‘‘˜ptolemy/doc/main/sectionMake’’ – using a link rather than a copy guarantees that all sections will print the same even if the master copy is modified.

7.3.3. Index entries

The documentation scripts for **Ptolemy** will automatically generate an index. The indexing system supports three kinds of index entries: ‘‘definition’’ entries are used when a term is defined in some sense; ‘‘example’’ entries are used when a term is used in an example; and ‘‘reference’’ enteries are used in all other cases – typically when a term is referenced. When the index is printed, definition entries have their page numbers in boldface, index entries have their page numbers in italics, and reference entries have their page number in the default font. This allows the reader to quickly locate the information for which they are searching.

Entries in the index must be defined in the document source using one of the the macros:

```
.Id "definition entry"
.Ie "example entry"
.Ir "reference entry"
```

for a definition, example, or reference entry, respectively. (The .IE macro used in older versions of Ptolemy acts as a reference entry). The entries will be sorted alphabetically. If you look at the index printed with this document, you will find an entry ‘‘index’’ pointing right here. If you wish for a two word entry to appear twice, alphabetically by each of the two words, then you must use two index entries:

```
.Ir "two words"
.Ir "words, two"
```

No additional editing will be done on index entries. For this reason, index entries should be carefully chosen. For example, please do not put the following entry in every demo description:

```
.IE "Ptolemy demo"
```

This will lead to hundreds of identical, and very uninteresting index entries. Avoid index entries beginning with very generic words in the **Ptolemy** vocabulary, like **Ptolemy**, `star,` `galaxy,` or `domain.` Of course, if you are writing some explanation of these basic terms, then an index entry is appropriate. Before entering index entries for a star, look in the documentation for similar stars to get an idea of the subject terms that have already been used and might be related.

One final note: to generate an index entry for a star or galaxy, use one of the macros:

```
.Sr star domain
.Se star domain
```

These will create reference or example index entry for the named star in the given domain in the ‘‘C++ class font’’ (to set it off from other entries), and will parenthetically note that the entry is for a star in the specified domain. If the star is in the same domain as the current section of the documentation, you can leave off the domain specifier. For example, the entry:

```
.Sr autocor
```

when in SDF domain documentation will generate an index entry of the form:

`autocor` (SDF star)

7.3.4. Special fonts and displays

By convention, **Ptolemy** documentation always refers to **Ptolemy** using the troff string invoked by the sequence ‘‘*(PT’’. This gets formatted in a standard way with a standard font. The overall **Ptolemy** version number is 0.4, invoked by the sequence ‘‘*(PV’’.

In addition, c++ class names are set in a special font by using the ‘‘.c’’ macro:

```
.c Class
```

as shown here: **`Class.`** For a display entirely set in this font, with no fill, use the macro pair ‘‘.(c ... .)c’’, as in

```
.(c
text
goes
here.
.)c
```

which will come out like:

```
text
goes
here.
```

This font is also used to highlight keywords, and to display sections of source code. The ‘‘.(c ... .)c’’ macro pair attempt to keep the text between them on one page, starting a new page if there isn’t room on the current one. The ‘‘.(d ... .)d’’ macro pair have the same function but without the keep.

7.3.5. Documenting stars

Manual entries for new stars will be generated automatically if you follow the directions given in the section above on defining new stars. You need not, therefore, be concerned with the troff macros used for this. All editing of the star documentation should be done in the .pl file, not in the troff file.

7.3.6. Documenting new galaxies and universes

Unlike stars, galaxies and universes must be documented manually. The troff source files for galaxies should go in XXX/doc/stars, where XXX is the location of the domain source directory. Since galaxies can be used exactly as if they were stars, it is logical that their documentation should appear with the stars. Troff source files for demo universes should go in XXX/doc/demos, where XXX is again the location of the domain source directory.

In both cases, the syntax of the document source files is as follows:

```
.\" %W%   %G%
.NA "Name of demo universe"
.SD
Short descriptor
.SE
.SV %I% %G%
.AL "Author list, with commas, and a period."
.LO "Location of the galaxy or demo definition"
.LD
Long descriptor.
This can contain any -me, eqn, or tbl directives.
It can also contain un-numbered headers:
.UH "title"
.SA
"See Also"
List of pointers to other entries in the manual
.ES
```

The first line is again a *ditroff* comment, containing the SCCS version ID. The second line defines how the version ID and date should be printed when the manual is printed. The version number %I% and date %G% keywords have the same meaning as before. The date here must be manually modified when the galaxy or demo is modified. The rest of the entries should be self-explanatory. Here is an example:

```
.\" %W%   %G%
.NA adaptFilter
.SD
A demo of the LMS adaptive filter star
.DE
.Se LMS
.SV %I% %G%
.AL "Edward A. Lee"
.LO "~ptolemy/src/domains/sdf/demo"
.LD
A Gaussian white noise signal serves as input to an
FIR filter and an adaptive filter using the LMS algorithm.
The output of the two filters and the error are displayed
after the run.  By examining the parameters of the two
filters, you can find the filenames for the FIR filter
coefficients and the initial tap values.  The final tap
values can be saved by specifying a saveTapsFile in the
adaptive filter star.
.SA
IIDGaussian,
LMS,
FIR.
.ES
```

Note that a carefully chosen index entry has been inserted here via the ‘‘.Se’’ macro. *It is not necessary to insert an explicit index entry for the name of the galaxy or demo.* This will be done automatically. It is also recommended that index entries not be inserted for all the ‘‘See Also’’ entries. In this case, however, the main purpose of the demo is to demonstrate the **LMS** star, and the name of the demo is not something like **LMSdemo,** so an explicit entry is advised.

For galaxies, additional entries are required to document the inputs, outputs, and states. The list of inputs is surrounded by the delimitters

```
.IH
...
.PE
```

while the list of outputs is inside

```
.OH
...
.PE
```

Each input or output is listed as follows

```
.NE
\fIname\fR (type): descriptor
```

States are delimitted by

```
.SH
...
.ET
```

with each entry reading

```
.NE
\fIname\fR (type): descriptor
.DF "default value"
```

State entries should also be made for demos with top-level parameters.

A single star, galaxy, or demo document page can be printed using the command ‘‘printpage filename’’. This command is stored in ˜ptolemy/doc/bin, but also has a link in ˜ptolemy/bin, so it should be in your path.

7.3.7. Documenting targets

Targets should be documented in the same way that universes and galaxies are documented, with careful attention paid to documenting each parameter or state. Troff source files for demos should go in XXX/doc/targets, where XXX is again the location of the domain source directory.

7.3.8. Documenting new domains

To include a new domain in the manual, you must edit the Makefile in ˜ptolemy/doc to add to the `DIRS` variable the name of the directory containing the domain documentation. This will cause your domain to be printed along with the rest of the manual. Suppose your domain is documented in directory XXX/doc. This directory should contain a set of specific files recognized by the documentation scripts.

title: The directory XXX/doc should contain a file called ‘‘title’’ which defines the title page for the domain. A typical title page is:

```
.\" %W%   %G%
.eh ''''
.oh ''''
.bp
.(b C
.sp 1.5i
.S 200
.ft 6
2
.S 80
.sp 2i
SDF
.sp 1i
Domain
.)b
```

These are all -me macro calls, except ‘‘.S’’ which is defined in ˜ptolemy/doc/headers/shared. It changes the font size in an orderly way. The eighth line from the top specifies a section number for the overall **Ptolemy** document. This is optional, but should obviously be a large enough number so that is doesn’t conflict with other domain sections.

init: The directory XXX/doc should contain a file called ‘‘init’’ with the following definitions:

```
.\" %W%   %G%
.nr SN 0
.nr AS 2
```

where the section number is changed to be the same as that used in the ‘‘title’’ file. Pages of the domain section will be numbered X-Y, where X is the section number defined by the init file, and Y is the page number.

overall: Just as with ordinary manual sections, this file contains the main documentation of the domain, including explanations of its basic operation, how to add stars, how its interface to other domains works, etc. The macros it uses are identical to those of ‘‘basic documents’’ as explained above.

stars: This directory should contain the troff source documents for the stars in the domain. If the directory is absent, no stars documentation will be printed.

starstitle: This file contains the title of the stars documentation section, as in the following example:

```
.\" %W%   %G%
.TI "SDF Stars"
```

demos: This directory should contain the troff source documents for the demos in the domain. If the directory is absent, no demos documentation will be printed.

demotitle: This file contains the title of the demos documentation section, as in the following example:

```
.\" %W%   %G%
.TI "SDF Demos"
```

7.3.1. Restricted filenames in domain documents

The following filenames in an XXX/doc directory, where XXX is the domain directory, have special functions, as explained above:

```
index
title
init
overall
stars (a directory)
starstitle
demos (a directory)
demostitle
Makefile
```

Complete documentation of a domain contains all of these, plus possibly other files sourced using .so.

7.4. Configuring your system to use the Gnu groff program

If you do not have *ditroff*, you may preview and print the documents by configuring your system to use *groff* instead. The *groff* package is a set of tools from the Free Software Foundation that provide a freely redistributable, mostly-compatible replacement for *troff*, *nroff*, and related programs.

Since the command line interface to *groff* is quite different and some adjustments are required, different scripts are provided. To install them as the official document-printing scripts, do the following (you must have appropriate write permission):

```
% cd ~ptolemy/doc/bin
% USE_GROFF
```

This command will install the *groff* scripts as the official scripts.

Since *groff* is a replacement for the old *troff* rather than for *ditroff*, it cannot handle as many fonts; therefore the scripts for using *groff* cause some of the fancier fonts used in the manuals to print in bold face instead. There may be other glitches as well; however, the resulting manuals are still quite usable.

The *groff* program can be obtained from many FTP sites; the Free Software Foundation will also send a copy of it to you on tape for a reasonable fee. Write to:
Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA
or send email to ‘‘gnu@prep.ai.mit.edu’’ for more information on obtaining FSF software.

8. xgraph — the plotting program

Authors:

David Harrison

8.1. Introduction

The *xgraph* program draws a graph on an X display given data read from either data files or from standard input if no files are specified. In **Ptolemy**, this program is invoked by several stars in several domains, and by the plot command in Pigi. The program is also available for stand-alone use, independent of **Ptolemy**. Xgraph can display up to 64 independent data sets using different colors and/or line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels, and a legend. There are options to control the appearance of most components of the graph.

8.2. Invoking xgraph

The synopsis for stand-alone invocation of xgraph is

xgraph [options] [=WxH+X+Y] [-display host:display.screen] [file ...]

The options are explained below. When invoking xgraph through a **Ptolemy** star, or the plot command, any of the options can be specified. Hence the full flexibility of the program is available to the user.

8.3. Detailed description

The input format is similar to *graph(1G)* but differs slightly. The data consists of a number of *data sets.* Data sets are separated by a blank line. A new data set is also assumed at the start of each input file. A data set consists of an ordered list of points of the form "{directive} X Y". The directive is either "draw" or "move" and can be omitted. If the directive is "draw", a line will be drawn between the previous point and the current point (if a line graph is chosen). Specifying a "move" directive tells xgraph not to draw a line between the points. If the directive is omitted, "draw" is assumed for all points in a data set except the first point where "move" is assumed. The "move" directive is used most often to allow discontinuous data in a data set. The name of a data set can be specified by enclosing the name in double quotes on a line by itself in the body of the data set. The trailing double quote is optional. Overall graphing options for the graph can be specified in data files by writing lines of the form "<option>: <value>". The option names are the same as those used for specifying X resources (see below). The option and value must be separated by at bleast one space. An example input file with three data sets is shown below. Note that set three is not named, set two has discontinuous data, and the title of the graph is specified near the top of the file.

```
TitleText: Sample Data
0.5 7.8
1.0 6.2
"set one
1.5 8.9
```

```
"set two"
-3.4 1.4e-3
-2.0 1.9e-2
move -1.0 2.0e-2
-0.65 2.2e-4

2.2 12.8
2.4 -3.3
2.6 -32.2
2.8 -10.3
```

After *xgraph* has read the data, it will create a new window to graphically display the data. The interface used to specify the size and location of this window depends on the window manager currently in use. Refer to the reference manual of the window manager for details.

Once the window has been opened, all of the data sets will be displayed graphically (subject to the options explained below) with a legend in the upper right corner of the screen. To zoom in on a portion of the graph, depress a mouse button in the window and sweep out a region. *xgraph* will then open a new window looking at just that portion of the graph. *xgraph* also presents three control buttons in the upper left corner of each window: *Close*, *Hardcopy*, and *About.* Windows are closed by depressing a mouse button while the mouse cursor is inside the *Close* button. Typing EOF (control-D) in a window also closes that window. Depressing a mouse button while the mouse cursor is in the *Hardcopy* button causes a dialog to appear asking about hardcopy (printout) options. These options are described below:

Output Device

> Specifies the type of the output device (e.g. "HPGL", "Postscript", etc). An output device is chosen by depressing the mouse inside its name. The default values of other fields will change when you select a different output device.

Disposition

> Specifies whether the output should go directly to a device or to a file. Again, the default values of other fields will change when you select a different disposition.

File or Device Name

> If the disposition is "To Device", this field specifies the name. A device name is the same as the name given for the -P command of lpr(1). If the disposition is "To File", this field specifies the name of the output file.

Maximum Dimension

> This specifies the maximum size of the plot on the hardcopy device in centimeters. *xgraph* takes in account the aspect ratio of the plot on the screen and will scale the plot so that the longer side of the plot is no more than the value of this parameter. If the device supports it, the plot may also be rotated on the page based on the value of the maximum dimension.

Include in Document

> If selected, this option causes *xgraph* to produce harcopy output that is suitable for inclusion in other larger documents. As an example, when this option is selected the Postscript output produced by xgraph will have a bounding box suitable for use with psfig.

Title Font Family

This field specifies the name of a font to use when drawing the graph title. Suitable defaults are initially chosen for any given hardcopy device. The value of this field is hardware specific -- refer to the device reference manual for details.

Title Font Size

This field specifies the desired size of the title fonts in points (1/72 of an inch). If the device supports scalable fonts, the font will be scaled to this size.

Axis Font Family and Axis Font Size

These fields are like *Title Font Family* and *Title Font Size* except they specify values for the font *xgraph* uses to draw axis labels, and legend descriptions.

Control Buttons

After specifing the parameters for the plot, the "Ok" button causes *xgraph* to produce a hardcopy. Pressing the "Cancel" button will abort the hardcopy operation. Depressing the *About* button causes Xgraph to display a window containing the version of the program and an electronic mailing address for the author for comments and suggestions.

8.4. Options

Xgraph accepts a large number of options most of which can be specified either on the command line, in the user's .Xdefaults or .Xresources file, or in the data files themselves. A list of these options is given below. The command line option is specified first with its X default or data file name (if any) in parenthesis afterward. The format of the option in the X defaults file is "program.option: value" where program is the program name (xgraph) and the option name is the one specified below. Option specifications in the data file are similar to the X defaults file specification except the program name is omitted.

=WxH+X+Y (Geometry)

Specifies the initial size and location of the xgraph window.

–<digit> <name>

These options specify the data set name for the corresponding data set. The digit should be in the range ’0’ to ’63’. This name will be used in the legend.

–bar (BarGraph)

Specifies that vertical bars should be drawn from the data points to a base point which can be specified with -brb. Usually, the -nl flag is used with this option. The point itself is located at the center of the bar.

–bb (BoundBox)

Draw a bounding box around the data region. This is very useful if you prefer to see tick marks rather than grid lines (see -tk).

–bd <color> (Border)

This specifies the border color of the *xgraph* window.

–bg <color> (Background)

Background color of the *xgraph* window.

–brb <base> (BarBase)

This specifies the base for a bar graph. By default, the base is zero.

−brw <width> (BarWidth)

This specifies the width of bars in a bar graph. The amount is specified in the user's units. By default, a bar one pixel wide is drawn.

−bw <size> (BorderSize)

Border width (in pixels) of the *xgraph* window.

−db (Debug)

Causes xgraph to run in synchronous mode and prints out the values of all known defaults.

−fg <color> (Foreground)

Foreground color. This color is used to draw all text and the normal grid lines in the window.

−gw (GridSize)

Width, in pixels, of normal grid lines.

−gs (GridStyle)

Line style pattern of normal grid lines.

−lf <fontname> (LabelFont)

Label font. All axis labels and grid labels are drawn using this font. A font name may be specified exactly (e.g. "9x15" or "-*-courier-bold-r-normal-*-140-*") or in an abbreviated form: <family>-<size>. The family is the family name (like helvetica) and the size is the font size in points (like 12). The default for this parameter is "helvetica-12".

−lnx (LogX)

Specifies a logarithmic X axis. Grid labels represent powers of ten.

−lny (LogY)

Specifies a logarithmic Y axis. Grid labels represent powers of ten.

−lw width (LineWidth)

Specifies the width of the data lines in pixels. The default is zero.

−lx <xl,xh> (XLowLimit, XHighLimit)

This option limits the range of the X axis to the specified interval. This (along with -ly) can be used to "zoom in" on a particularly interesting portion of a larger graph.

−ly <yl,yh> (YLowLimit, YHighLimit)

This option limits the range of the Y axis to the specified interval.

−m (Markers)

Mark each data point with a distinctive marker. There are eight distinctive markers used by xgraph. These markers are assigned uniquely to each different line style on black and white machines and varies with each color on color machines.

−M (StyleMarkers)

Similar to -m but markers are assigned uniquely to each eight consecutive data sets (this corresponds to each different line style on color machines).

−nl (NoLines)

Turn off drawing lines. When used with -m, -M, -p, or -P this can be used to produce scatter plots. When used with -bar, it can be used to produce standard bar graphs.

−p (PixelMarkers)

Marks each data point with a small marker (pixel sized). This is usually used with the -nl option for scatter plots.

–P (LargePixels)
Similar to -p but marks each pixel with a large dot.

–rv (ReverseVideo)
Reverse video. On black and white displays, this will invert the foreground and background colors. The behaviour on color displays is undefined.

–t <string> (TitleText)
Title of the plot. This string is centered at the top of the graph.

–tf <fontname> (TitleFont)
Title font. This is the name of the font to use for the graph title. A font name may be specified exactly (e.g. "9x15" or "-*-courier-bold-r-normal-*-140-*") or in an abbreviated form: <family>-<size>. The family is the family name (like helvetica) and the size is the font size in points (like 12). The default for this parameter is "helvetica-18".

–tk (Ticks)
This option causes *xgraph* to draw tick marks rather than full grid lines. The -bb option is also useful when viewing graphs with tick marks only.

–x <unitname> (XUnitText)
This is the unit name for the X axis. Its default is "X".

–y <unitname> (YUnitText)
This is the unit name for the Y axis. Its default is "Y".

–zg <color> (ZeroColor)
This is the color used to draw the zero grid line.

–zw <width> (ZeroWidth)
This is the width of the zero grid line in pixels.

Some options can only be specified in the X defaults file or in the data files. These options are described below:

<digit>.Color
Specifies the color for a data set. Eight independent colors can be specified. Thus, the digit should be between ’0’ and ’7’. If there are more than eight data sets, the colors will repeat but with a new line style (see below).

<digit>.Style
Specifies the line style for a data set. A string of ones and zeros specifies the pattern used for the line style. Eight independent line styles can be specified. Thus, the digit should be between ’0’ and ’7’. If there are more than eight data sets, these styles will be reused. On color workstations, one line style is used for each of eight colors. Thus, 64 unique data sets can be displayed.

Device
The default output form presented in the hardcopy dialog (i.e. "Postscript", "HPGL", etc).

Disposition
The default setting of whether output goes directly to a device or to a file. This must be one of the strings "To File" or "To Device".

FileOrDev
The default file name or device string in the hardcopy dialog.

ZeroWidth

Width, in pixels, of the zero grid line.

ZeroStyle

Line style pattern of the zero grid line.

8.5. Bugs

- Zooming in on bar graphs doesn’t work right.
- There is no way to produce hardcopy without running xgraph interactively.

9. Filter design programs

Authors:

E. A. Lee

9.1. Introduction

Pending the inclusion of more sophisticated filter design software with **Ptolemy**, this distribution includes two Fortran programs for this purpose. These design FIR filters using the Parks-McClellan algorithm or the window method. The frequency sampling method can be done directly using **Ptolemy** stars, as shown in the demos included with the SDF domain.

These programs can be invoked standalone or through the filter command in pigi. If invoked through pigi, then an xterm window will be opened and the program started.

Filter specifications can be entered by hand or loaded from a file. The first prompt from the program is the name of the input command file. Simply typing return will result in manual entry. To enter data from a file, the data should appear in the file in exactly the order that it would appear if it were being entered by hand, with one question answered per line. Unfortunately, these command files are rather difficult to read, and not too easy to create correctly. It is recommended to first do manual entry, then imitate the entries in a file. Data can be entered in any reasonable numeric format.

9.1.1. Caveats

These are public domain Fortran program, provided for convenience, although they are not an integral part of **Ptolemy**. Incorrect formatting of data line can lead to ungraceful exits (often with a core dump) or incorrect results. Either program can be invoked through the **Ptolemy** graphical interface, in which case it is started in its own window, in the background. Note that in this case the program always starts in your home directory.

9.2. optfir — equirriple FIR filter design

Optfir is a Fortran program performing classical FIR filter design using the Parks-McClellan algorithm. There is nothing unusual about this program, so almost any DSP book will give an adequate explanation of the algorithm as well as the meaning of its arguments. Briefly, the program permits the design of bandpass filters, differentiators, hilbert transformers, and half-band filters.

Bandpass filters include lowpass, highpass, bandstop, and multiband, where each band can have a different gain (the "desired value"). A desired value of 0 specifies a stopband. The weight associated with each band determines the ratio of ripple in each band; a higher weight means less ripple.

9.3. wfir — window method FIR filter design

Wfir is a Fortran program performing classical FIR filter design using the window method. There is nothing unusual about this program, so almost any DSP book will give an adequate explanation of the algorithm as well as the meaning of its arguments. Briefly, the program permits the design of lowpass, highpass, bandpass, and bandstop filters using any of a number of windows. The method is to first compute the impulse response of an ideal (brick wall) filter, and then window it with the selected window to make the impulse response finite.

10. VEM - Interactive Graphics for Oct Designs

Authors:

David Harrison
(modified by Edward A. Lee)

10.1. Introduction

VEM is a graphical editor for designs that are stored using the Oct design manager. The **Ptolemy** interactive graphical interface (pigi) uses VEM. A process called *pigiRpc* runs as a VEM application. Most users can get all the information they need to run pigi from the pigi section of the Almagest. This section contains more extensive documentation of VEM, and is extracted from standard documentation for the octtools distribution. No Oct documentation is included. This is required only by programmers wishing to modify pigiRpc, and is available from the Industrial Liaison Program office, Dept. of EECS, UC Berkeley, Berkeley CA 94720.

10.2. VEM (version 8.1)

VEM is an interactive graphics shell/editor for designs represented in the **Oct** CAD/VLSI data representation system. The primary purpose of VEM is to provide a means of looking at the graphics representation of **Oct** views and invoking various CAD tools on these views. **Ptolemy** is such a CAD tool, invoked via a program called *pigiRpc*. VEM was originally intended for IC design. As such VEM provides standard graphics editing capabilities for physical (mask-level), symbolic, and schematic designs. **Ptolemy** uses schematic capabilities for applications and physical capabilities for icons. Some understanding of the terminology of the **Oct** data manager is required to use VEM. This terminology is explained in the pigi section of the Almagest.

VEM is a multi-window application designed to run on bitmap based graphics workstations. VEM will run on any such system that has a reasonably complete implementation of the *X* Window System version 11 release 4 or later. Currently, there are *X* implementations for a wide range of desktop workstations including all DEC and Sun Microsystems computers as well as some HP, Apollo, and Masscomp computers. Currently, **Ptolemy** has only been tested on DEC and Sun Miscrosystems computers.

VEM may be started by simply typing "vem", but this will not start **Ptolemy**. To start the **Ptolemy** interactive graphical interface, simply execute the command pigi in ˜ptolemy/bin. This invokes a shell script that starts VEM and the associated pigiRpc process. VEM is started in general with the following command line options:

```
vem [-F cell[:view[:facet]]] [-G WxH+X+Y] [-R [host,]path] \
     [-display host:display] [name=value ...]
```

As an example, the main part of the pigi script reads:

```
if (!($?PIGIRPC)) then
     setenv PIGIRPC ~ptolemy/bin.$ARCH/pigiRpc
endif
xrdb -m ~ptolemy/lib/pigiXRes8
vem -G 600x150+0+0 -F init.pal:schematic \
     -G 600x300+0+170 -R $PIGIRPC
```

The first three lines set the PIGIRPC environment variable to point to the correct executable. The next line merges the X Windows resources defined in ~ptolemy/lib/pigiXRes8. The last line starts VEM.

VEM looks at the value of the DISPLAY environment variable to determine what host and display to use for output. VEM and pigi may be run in the background without affecting the program operation.

The **-F, -G,** and **-R** command line options allow a user to specify a certain startup window configuration for VEM. These three options are considered triplets that specify the initial cell, position and size, and remote application respectively for a window. There is no limit to the number of triplets that may be specified. The **-F** flag marks the start of each new triplet. The corresponding **-G** and **-R** flags after the **-F** flag are optional. If the **-G** flag is omitted, VEM will not specify a location for the window and most window managers will interactively prompt for the window location. If the **-R** flag is omitted, no remote application will be started in the window. The **-F** flag can be omitted from the first triplet. In this case, the **-G** and **-R** flags apply to the console window. For example, the pigi script above starts VEM with its console window at (0,0) with a size of 600 by 150, and one window looking at the cell "init.pal:schematic" at (0,170) with a size of 600 by 300, running the "pigiRpc" remote application.

VEM is a highly customizable editor. Nearly all of the colors, font styles and fill patterns VEM uses can be changed by the user. Normally, these parameters are read from the user's X resources (usually kept in the files ~/.Xdefaults or ~/.Xresources). However, one can set certain parameters on the command line using the = (equal) command line option. A list of all configurable parameters can be found in the section "Customizing VEM".

If no location for the console window is provided on the command line, VEM will not specify a location for the console window and most window managers will interactively prompt for a console window location. This window echoes user input and outputs various help and status messages. After starting, the console will display a prompt and wait for input.

If the init.pal facet does not exist in the directory in which pigi is started, then it will be created. A blank cell will appear. Convention in **Ptolemy** dictates that this facet should be used to store icons representing complete applications, or universes, that are defined in the directory. If such icons already exist in the init.pal facet, the applications can be examined using the pigi "look-inside" command.

New windows can be created using *open-facet* command (see the pigi command reference). Note that these windows will be connected to the pigiRpc process. It is also possible to open a window from the VEM console using the *open-window* command, but this new window will not be attached to pigiRpc (see the command reference below). Each window has exactly one associated cell. Mouse action with the cursor positioned inside a window cause operations to occur to the window's associated cell. Any number of windows can be created with the same or different associated cells. More than one window may have the same associated cell. In this case, all of the windows are attached to the *same* cell. Thus, a change to one of the windows may cause updates to other windows which look at the same cell.

VEM assumes the presence of a three-button mouse. The left button is used for entry of graphics information. The middle button is used for the primary menu of commands. The right button is used to modify graphics information entered using the left button.

Commands to VEM are specified in post-fix form. The user builds an argument list first and then selects a command. Commands can be selected in four ways: pop-up menus, single keystrokes, using the last command button, or by typing in the command name. Pressing and releasing the middle button in a graphics window causes a VEM menu to appear. The user can use the mouse to riffle through the options until the desired choice is highlighted. Pressing and releasing the mouse button activates the selected command. Pressing and releasing the mouse outside the menu cancels the selection. Normally, pressing and releasing the middle button causes a VEM menu to appear. Holding the shift key and clicking the middle button causes the pigi menu to appear. Both menus are useful.

A number of common commands can be selected via a single keystroke. Key bindings for various commands are shown next to the corresponding entry in the VEM menu, are listed in the command reference below, and can be queried interactively using the *bindings* command. Finally, typing a colon (:) allows the user to type in the command name (or a user defined alias). The standard line editing keys can be used while typing the command name.

The interface supports automatic command completion. Typing a tab will complete the command if it is unique or offer a list of alternatives if it is not unique. The command is selected by typing a carriage return.

There are five types of input to VEM: points, boxes, lines, text, and objects. Points are entered by pressing and releasing the left button of the mouse. Boxes are entered by pressing, dragging, and releasing the left button. Lines are entered by pressing, dragging and releasing the left button over a previously created point or line. Text is entered by typing the text enclosed in double quotes. If entering a filename, typing a TAB character will cause VEM to try to complete the name if it is unique or offer a list of alternatives if it is not unique. Objects are entered using the *select-objects* and *unselect-objects* commands. The last item on an argument list can be deleted using the standard character for delete. The last group of items can be deleted using the word erase character control-W. The entire argument list can be deleted using the standard kill-line character (usually control-X or control-U).

Once entered, graphics arguments (points, boxes, and lines) can be modified in various ways. For all arguments, a point can be moved by moving the cursor over the point and pressing, dragging, and releasing the right mouse button. New points can be added to a group of lines by moving over a point in the segment, depressing, dragging, and releasing the left mouse button. This will insert a new point after the point. It is also possible to interactively move, rotate, and mirror objects arguments (selected items). See the description of the *transform* command in the command reference below.

This version of VEM supports three basic editing styles: physical, symbolic, and schematic. Physical editing involves the entry and editing of basic geometry and the creation of interface terminals. This style is used in pigi to build icons. Symbolic editing involves the placement of instances of leaf cells and the interconnection of these instances. Pigi does not use symbolic editing. Schematic editing is an extension of symbolic where the primitive cells are schematic symbols and wire width is insignificant. Schematic cells use used by pigi to represent block diagrams.

When VEM opens a new cell, a dialog will appear asking for three property values: TECHNOLOGY, VIEWTYPE, and EDITSTYLE. The TECHNOLOGY and VIEWTYPE properties determine the location of the technology facet (see tap(3)). A standard technology facet has been designed for **Ptolemy**, so the defaults that appear are almost always acceptable. Layer display and design rule information is read from this facet. The EDITSTYLE property is used by VEM to determine the set of commands available for editing the cell. Currently, the legal editing styles are PHYSICAL, SYMBOLIC, or SCHEMATIC.

10.2.1. Using Dialog Boxes

Some commands require information that cannot be expressed easily using post-fix notation. Examples include destructive commands that require an explicit confirmation and commands that require complex non-graphic information. VEM uses *dialog boxes* to handle these situations. Dialog boxes are windows that resemble business forms. These windows contain labeled fields for entering text, changing numerical values, and selecting options. This section describes how to use dialog boxes.

All dialog boxes in VEM have the same form. At the top of all dialogs is a one line title indicating the purpose for the dialog. The middle of the dialog (known as the body) contains fields for displaying and editing information of various kinds. At the bottom of the dialog are a number of *control buttons*. Each control button represents a command. The arguments to the command are the values of the fields displayed in the body. Thus, operating a dialog consists of editing or changing fields in the body and then selecting a command by activating a control button. Six kinds of fields may appear in the body of a VEM dialog: editable text, non-editable text, enumerated value, numerical value, exclusive lists, and non-exclusive lists. A description of each field type is given in the paragraphs that follow.

Editable text fields are used to enter and edit text. Visually, an edit text field consists of a box containing a caret cursor, an optional scrollbar, and a label to the left of the box indicating the purpose for the field. Only one editable text field is active in any one dialog. The active editable text field has a dark border. Typing text with the mouse positioned anywhere in the dialog inserts text into the active editable text field. Most of the basic emacs editing commands can be used to modify the text in the field:

Key	**Action**	**Key**	**Action**
Ctrl-A	Beginning of Line	Ctrl-E	End of Line
Ctrl-B	Backward character	Ctrl-F	Forward character
Ctrl-N	Next line	Ctrl-P	Previous line
Ctrl-V	Next page	Meta-V	Previous page
Ctrl-D	Delete next character	Meta-D	Delete next word
Ctrl-K	Delete to end of line	Ctrl-S	Search forward
Ctrl-Y	Yank deleted text	Meta-I	Include file

The insert position in the field may also be changed by pressing the left mouse button when the mouse cursor is over the desired position. Any editable text field can be made active by clicking the left mouse button inside the editable area. Alternatively, one can use the <Tab> key to make the next text field active and <Meta-Tab> to make the previous field active. Editable text fields that display large amounts of text have a scrollbar to the left of the text area. Pressing the left and right mouse buttons when the mouse cursor is in a scrollbar will scroll the text down and up respectively in proportion to the distance between the mouse cursor and the top of the scrollbar. As an example, pressing the left mouse button near the bottom of the scrollbar will scroll down the text almost one screen. Pressing and releasing the middle mouse button scrolls the text to a relative position based on how far the mouse cursor is from the top of the scrollbar. Holding down the middle mouse button will interactively scroll through the text.

Non-editable text fields are used to display text messages. They consist of a box containing text and an optional scrollbar. The scrollbar operates just like those used in editable text fields.

Enumerated value fields are used to specify one value out of a small list of values. They consist of a value displayed inside a box and a descriptive label to the left of the value. The border of the value highlights as the mouse cursor moves over it. Depressing and holding the left mouse button inside the value box causes a menu to appear that displays all possible values. The choices will highlight as the mouse cursor moves over them. To select a new value, release the mouse button when the desired choice is highlighted. The new value will appear in the value box. One can leave the value unchanged by releasing the mouse button outside the menu boundary.

The numerical value field is used to specify a magnitude between a predetermined minimum and maximum. Visually, it consists of a box containing a numerical value, a horizontal scrollbar to the right of the box for changing the value, and a label to the left describing the value. The magnitude of the value is changed by operating the scrollbar. Pressing the left and right buttons in the scrollbar decrement and increment the value by one unit. Pressing the middle button changes the value based on the distance between the mouse cursor and the left edge of the scroll bar. The middle button may be pressed and held to interactively modify the value. Most people use the middle button to set the value roughly then use the left and right buttons to make the value precise.

Exclusive lists are used to choose one possible value out of a (possibly quite large) list of values. These values are displayed in a box with a scrollbar on the left edge of the box. Each value consists of a button box on the left and a descriptive label to the right. As the mouse moves over a button box, it will highlight to indicate it can be activated. The button box of the selected item will appear dark while all others will remain light. If there are too many values to display in the box at one time, the scrollbar can be used to scroll through the possible values. The scrollbar operates in the same way as described for editable text fields.

Non-exclusive lists are used to choose zero or more possible values from a (possibly quite large) list of values. A non-exclusive list resembles an exclusive list both in appearance and operation. However, unlike an exclusive list, one can choose any number of items in a non-exclusive list. Visually, the two lists are distinguished by the appearance of the button boxes. Exclusive button boxes resemble radio buttons. Non-exclusive button boxes resemble check marks. Pressing the left button in a non-exclusive button box causes the value to toggle (i.e., if it was selected it becomes unselected, if it was unselected it becomes selected).

Control buttons cause the dialog to carry out some operation. They consist of a text label surrounded by a box. Control buttons are activated in one of two ways: pressing and releasing the left mouse button when the mouse cursor is positioned inside the button boundary, and through keystrokes. Not all control buttons can be activated using keystrokes. Those that can be activated in this fashion display the key in parentheses under the button label. Although there are exceptions, most dialogs support the following keyboard accelerators:

Keys	Action
<Ret>,<Meta-Ret>,<F1>	Ok
<Del>,<Meta-Del>,<F2>	Cancel
<Help>,<F3>	Help
<F4>	All
<F5>	Clear

Dialogs may be both *moded* and *unmoded*. Moded dialogs are those requiring a response before processing can proceed. VEM uses these kinds of dialogs to ask for confirmation before proceeding. On the other hand, unmoded dialogs remain active until explicitly dismissed by the user. Other commands may be invoked freely while unmoded dialogs are visible. Most non-confirmation dialogs in VEM are unmoded.

10.2.2. General Commands

Below is a reference for all VEM commands. The first section outlines general commands available for editing all types of cells. The next section describes the general selection mechanism. The third section describes property and bag editing features. The next three sections describe editing commands for physical, symbolic, and schematic cells respectively. The summary includes the name of the command as it appears on a menu, if it appears in the menu. The command name can be typed in as well. Place the mouse in the window where you wish to execute the command, enter the command arguments (points, objects, etc.), type a colon ‘‘:’’, and type the command name. The TAB character will automatically complete command names. The phrase <no-name> implies it has no default menu or command name binding,

The list below also shows the default keyboard binding for each command, if it has one, and the syntax of the argument list passed to it. The symbol <*> implies the command has no default key binding. In general, the commands used most often have key and menu bindings. Less often used commands may have only command name bindings.

Some VEM commands are not documented here because they are dangerous or conflict with the objectives of pigi. Those commands will not appear in the VEM menu, and have no key binding, although all are still available by typing them in. Adventurous users may wish to consult the standard Octtools documentation before using them.

<no-name> <del>,ˆH Any Argument List

This command deletes the last item of the last argument on the argument list. Thus, if the last argument is 10 boxes, it will delete the last box entered and the argument list will be modified to contain 9 boxes.

<no-name> ˆW Any Argument List

This command is similar to the one above, but deletes the entire last argument on the argument list. Thus, if the last argument is 5 lines, it will delete all 5 lines and leave the remaining arguments unchanged.

<no-name> ˆX or ˆU Any Argument List

This command erases the entire argument list allowing the user to start over.

bindings b Saves Arguments

This command asks the user for a command and displays all of its current key, menu, and alias bindings. The command will display a prompt (vem bindings>) and the user can specify a command using any of the four means of normally specifying commands (via menu, single keystroke, type-in, or last command). The command also outputs a one line description of the command for help purposes.

close-window ^D No Arguments

The *close-window* command closes the window the cursor was in when the command was invoked. This DOES NOT flush the contents of the window to disk. Even after all windows looking at a facet are closed, the contents are not saved on disk. This must be done using the *save-window* or *save-all* commands.

deep-reread <*> [objs]

The *deep-reread* command is a specialized form of the re-read command. With no arguments, it re-reads a facet and all of master facets of its subcells (instances). Both the contents and interface facets of the instances are re-read. If a set of objects is specified, the command re-reads the master cells of the instances in the object set. Only the master cells of the instances are re-read; cells are not re-read recursively when using this form.

interrupt ^C No arguments

This routine interrupts (deactivates) the window containing the cursor. No drawing will be done in the window until a full redraw requested by the user (using pan, zoom, or redraw-window) is done. The key binding for this command can also be used while a window is drawing to immediately stop drawing in that window.

kill-buffer <*> "cell view {facet} {version}"

The *kill-buffer* command flushes a facet out of memory *without* saving its contents. If the string specification of the facet is missing, the facet is determined by the window containing the cursor. *All* windows looking at this facet are destroyed. There are no key or menu bindings for the command and it will ask the user for confirmation before carrying out the command.

log-bindings <*> Saves arguments

The *log-bindings* command writes out a description of all type-in, menu, and key bindings for all commands in the editing style of the window containing the cursor. This description is written to the log file for the session.

open-window o [box] or "cell:{view:{facet:{version}}}"

The *open-window* command is primary way to create new graphics windows in VEM. It takes a string specifying the cell to open. When specifying the cell portion of the name, typing a TAB will attempt to complete the string as a file or offer alternatives if the name is not unique. If this string is absent, it will duplicate the window containing the cursor. Normally, the extent of the duplicated window is the same as the parent window. However, if the user specifies a box, the duplicated window will be zoomed to that extent (see zoom-in). The string specifying the cell contains four fields. The last three are optional and default to "physical", "contents", and the null string respectively. It is possible to specify your own defaults for these fields. See the customization document for details. Newly created windows are always zoomed to contain all geometry in the cell.

If the cell does not exist, it will be created. When creating new cells, VEM prompts the user for required cell properties. See the introduction for details. Most of the time, the defaults presented in this dialog are acceptable and activating the *Ok* button is sufficient to proceed.

palette **P** **{"palette-name"}**

The *palette* command opens a new window onto a previously created facet which contains standard layers or instances for a given technology. This window can be used to select layers for creating geometry or instances for instantiation. The command takes one argument: the name of the palette. If omitted, it defaults to "layer".

Palette cells are found using the function tapGetPalette (see tap(3)). For all standard technologies and viewtypes, there is a "layer" palette. In the symbolic editing style, there are also "mosfet" and "connector" palettes which display mosfets and connectors respectively. In the schematic editing style, there are "device" and "gate" palettes which contain device level and gate level schematic primitives. New palettes can be added easily. See ‘‘Customizing VEM’’ for details.

pan **p** **[any arguments] [pnt]**

The *pan* command centers the window containing the cursor around the last point on the argument list. The window will be redrawn so that the argument list point is now the center of the window. The point need not be in the same window as the cursor. Thus, a user can point in a window showing a large portion of a cell and invoke the command in a more detailed window for a magnifying glass effect. If the point is omitted, the command assumes the cursor position is also the desired center point. This is the fast way to pan in a single window.

pop-context **)** **No Arguments**

This command pops off an input context from the context stack and replaces the current context with that context. See the *push-context* command for details.

push-context **(** **No arguments**

This command pushes the current argument list context onto the context stack and gives the user a new context. This can be used to do other commands while preserving entered arguments. Note that the current arguments remain displayed. The old context can be restored using the *pop-context* command. Four context levels are supported in the current version of VEM.

push-master **<*>** **{"facet"}**

The *push-master* command opens a new window on the master of the instance under the mouse cursor. This command can be used all editing styles. If a facet name is supplied, the command will use that facet name instead of "contents". In **Ptolemy**, this command is rarely needed. The *edit-icon* command accomplishes the same objective.

recover-facet **<*>** **No Arguments**

Unless directed otherwise, VEM saves all cells occasionally in case of a system crash or some other unforseen disaster. Whenever a new cell is opened, VEM checks to see if the last automatically saved version is more recent than the user saved version. If the automatically saved version is more recent, a warning is produced and the user version is loaded. One can use the *recover-facet* command to replace the cell with the more recent automatically saved version. The command displays a dialog containing a list of all of the saved versions. Generally, there are two possible alternative versions for a cell. The *autosave* version is written by vem automatically after a certain number of changes to the cell. The *crashsave* version is written when VEM detects a serious error. Note that the crashsave version may itself be corrupt since a serious error occurred just before it was written. This command is destructive: it replaces the cell with the selected alternate. One can use the *Cancel* button to abort the recovery. Before using the *recover-facet* command, it is often useful to view the alternate cells. This can be done by specifying the version (either "autosave" or "crashsave" as the last field in the cell specification to *open-window.*

re-read **<*>** **No Arguments**

The *re-read* command flushes the facet associated with the window containing the cursor out of memory and reads it back in from the disk. This can be used to see changes to a cell that were done outside VEM or to revert back to the cell before changes were made. This is a dangerous command and VEM asks for confirmation before proceeding.

redraw-window **ˆL** **[box]**

This command redraws the contents of the window containing the cursor. It does not effect the argument list (i.e. it can be done regardless of the argument list contents). If a box is provided, only the portion of the window in the box is redrawn. If the window is interrupted, this command will reactivate it. However, if the box form is used, VEM will only draw the specified area and leave the window deactivated. This can be used to selectively draw portions of a deactivated window.

same-scale **=** **[any arguments] [pnt]**

This command changes the scale of the window containing the mouse to the same scale as the window containing the last point on the argument list. It is commonly used to compare the sizes of two facets.

save-window **S** **[any arguments]**

This command saves the contents of the facet associated with the window where the command was invoked. It asks for confirmation and does not effect the argument list.

set-path-width **w** **[box] [line] [point] or ["size string"]**

This command sets the current path width for a given layer on a window-by-window basis. It takes one argument which may be points, lines, boxes, or text. For points and lines, the argument length must be two. The path width is set to the maximum manhattan distance between the points. For boxes, only one box is allowed at the new width is the larger of the two dimensions. For text, the string should contain the path width in lambda. If no width is specified, the path width will be set to the minimum layer width as specified in the technology.

The layer for the path width command is determined by looking at the object under the cursor. If there are objects on more than one layer under the cursor, a dialog will be presented and the user should choose one of the listed layers and press "OK" to continue.

show-all **f** **No Arguments**

The *show-all* command causes VEM to zoom the window containing the cursor so that all of the cell is displayed in the window. The key binding is an abbreviation for "full". It does not effect the argument list.

switch-facet **<*>** **["cell view {facet {version}}"] or [pnt]**

This command replaces the facet in the window containing the mouse with a different facet. The first form replaces the window's facet with the named facet. The second form replaces the window's facet with the facet of the window containing the point.

toggle-grid **g** **No Arguments**

The *toggle-grid* command toggles the visibility of the grid in the window containing the cursor.

version **V** **No arguments**

This command outputs the current version of Pigi to the console window.

where **?** **No arguments**

The *where* command can be used to find out the position of the cursor in terms of **Oct** units. It also displays a textual representation of the objects under the cursor. The command can be issued while building an argument list without effecting the list. Alternatively, if the argument list includes an object set, the where command will print textual descriptions of the selected items.

write-window **W** **"cell:view" [any arguments]**

The *write-window* command saves the contents of the cell associated with the window where the command was invoked under another name. This alternate name is specified by the "cell:view" argument.

zoom-in **z** **[any arguments] [box]**

The *zoom-in* command zooms the window containing the mouse to the extent indicated by the last box on the argument list. The box and the zoom window must be in the same facet. However, the extent may be in a different window from the mouse which can be used to achieve a magnifying glass effect. If the box is not provided, it zooms the window containing the mouse in by a factor of two. If provided, the command removes the box from the argument list but leaves other arguments untouched.

zoom-out **Z** **[any arguments] [box]**

The *zoom-out* command is the opposite of zoom-in. This command zooms the window containing the mouse out far enough so that the OLD contents of the window are contained in the extent of the box provided on the argument list. Thus, smaller boxes zoom out farther than larger ones. If the box is omitted, the window containing the mouse is zoomed out by a factor of two.

10.2.3. Options

The options commands below all post form windows which allow a user to change display parameters interactively. The default values for these parameters can be changed in your ˜/.Xdefaults file. See "Customizing VEM" for details.

All of the options dialogs below are *unmoded.* This means that the user can do other things while the dialog is posted. They will not go away until explicitly closed by the user. Details on the operation of dialog boxes can be found in the section *Using Dialog Boxes* at the start of the manual.

window-options <*> No arguments

This command posts a dialog which presents a number of window related options each of which can be modified by the user on a window-by-window basis. Two kinds of options are presented in this dialog: flag options and value options. Flag options have a check box on the left of the option and can be either on or off. Pressing the left mouse button in the check box toggles the value of the option. Value options display a numerical value in a box with a descriptive label to the left and a scrollbar to the right for changing the value. At the bottom of the dialog are four control buttons: *OK, Dismiss, Apply,* and *Help.* Pressing the left mouse button inside the *OK* button saves any options you may have changed and closes the window. The *Dismiss* button does not save any options and closes the window. The *Apply* button saves any changed options but does not close the window. The *Help* button will open a window containing a brief description of the dialog.

Each of the options and their meanings are given below:

Visible Grid

If set, a grid will be shown in the window.

Dotted Grid

If the grid is visible and this option is set, the grid will be drawn as dots rather than lines.

Manhattan Argument Entry

If this option is set, entering line arguments and dragging option sets will be restricted to manhattan angles. This option is set by default in the symbolic and schematic editing styles.

Argument Gravity

If set, all lines entered using the left button whose endpoints are near an actual terminal will be snapped to that terminal. This is especially useful when editing schematic diagrams (VEM automatically turns this option on when the edit style is set to schematic). The .Xdefault parameter *vem.gravity* specifies the maximum distance between line endpoints and terminals for gravity to have an effect (by default, 10 pixels).

Show Instance Bounding Boxes

If this option is set, VEM will display bounding boxes around all instances. The instance name will be displayed in the center of these bounding boxes.

Show Actual Terminals

When viewing very large designs, drawing the highlighting around actual terminals can be expensive. If this option is turned off, VEM will not draw highlighting around actual terminals.

Expand Instances

If on, the contents facet of instance masters will be displayed. Otherwise, the interface facet will be displayed. This has the same effect as the *toggle-expansion* command.

Visible Title Bar

If this option is set, VEM will display its own title bar above each graphics window. If the option is turned off, the title bar will be turned off.

Oct units per Lambda

This parameter specifies the number of **Oct** units per lambda. The output of the where command and the coordinate displays in the title bar are displayed according to the value of this parameter. By default, VEM uses 20 **Oct** units per lambda.

Snap

All graphic input into the window will be snapped to multiples of this parameter (given in **Oct** units). By default, VEM snaps to one lambda (20 **Oct** unit) intervals.

Major Grid Spacing

This parameter specifies the grid spacing of major grid lines in **Oct** units. If logarithmic grids are turned on, it specifies a multiplying factor for major grid line spacing (see Base below).

Minor Grid Spacing

This parameter specifies the grid spacing of minor grid lines in **Oct** units. If logarithmic grids are turned on, it specifies a multiplying factor for minor grid line spacing (see Base below).

Minimum Grid Threshold

This parameter specifies the smallest allowable space (in pixels) between grid lines before VEM stops drawing them.

Log Grid Base

If non-zero, this option selects a logarithmic grid. Normally, there are two grids drawn at a fixed number of **Oct** units (major and minor grid lines). In a logarithmic grid, grids are drawn at some constant (specified by the Major Grid and Minor Grid parameters above) times the nearest integral power of the grid base. For example, if the constant is two and the base 10, grids will be drawn at 2, 20, 200, etc.

Log Grid Minimum Base

This parameter specifies the smallest base to be used for drawing logarithmic grids (see above).

Solid Fill Threshold

This parameter specifies the size (in pixels) before a shape is drawn using solid fill rather than stipple fill. A large number specifies all geometry should be drawn solid regardless of its size.

BB Name Threshold

Label text drawn in bounding boxes (e.g. instances or terminals) may or may not be drawn depending on the size of the bounding box. This parameter specifies how many times wider the text may be than the box before the label is not drawn.

BB Abstraction Threshold

This parameter specifies the maximum size of a bounding box (in pixels) where it is acceptable to draw a filled box rather than an outline to speed the drawing process.

Contact Abstraction Threshold

This parameter specifies the maximum size of a contact (connector) where it is acceptable to draw an abstraction of the contact rather than its full definition. This can be used to tell VEM to draw contacts quickly.

Interface Facet

This string parameter specifies the name of the displayed interface facet. This facet will be used to draw instances in unexpanded mode.

layer-display **<*>** **No Arguments**

This command posts a dialog which can be used to selectively turn on or off the display of geometry on any layer. At the bottom of the dialog are six control buttons. The *Ok, Dismiss, Apply,* and *Help* buttons work in the same way as described for *window-options.* The *All* button automatically selects all of the layers displayed in the body of the dialog. The *Clear* button automatically unselects all of the layers displayed in the body of the dialog. Above the control buttons there is a list of all layers displayed in the window. The layers currently shown in the window have buttons to the left of the layer name that have check marks. Those that are not shown have buttons to the left of the layer name that appear empty. The state of a layer can be changed by moving the cursor over the corresponding button and depressing the left mouse button. Note that no window update will occur until either *Ok* or *Apply* are pressed.

10.2.4. Selection

The selection commands described below are used to manipulate object arguments on the argument list.

select-layer **.** **[objs] [pnts] [lines] [boxes] "layer"**

The *select-layer* command is similar to the *select-objects* command but allows the user to select only the geometries on a particular layer. This layer can be specified in two ways: the layer name can be typed in as the last argument or the command will try to determine the layer by looking at the geometry under the cursor when the command is invoked. If the spot for the layer is ambiguous, VEM will post a dialog presenting a choice between the layers.

select-objects **s** **[objs] [pnts] [lines] [boxes] "layer"**

The *select-objects* command is used for placing collections of objects on the argument list for further processing by other commands. It takes as arguments any number of points, lines, or boxes. Points select items under the point, lines select objects which cross the line, and boxes select objects inside the box. If *select-objects* is not given any point, line, or box arguments, it will try to select items under the cursor where the command was invoked. These semantics are described in detail below.

For each point, the command adds zero or more of the objects under the point to the list. If there is more than one object under the point, a dialog will be posted with buttons representing each of the objects under the point. Clicking the mouse in one of the buttons highlights the object. Once the user has clicked on the desired objects, the "ok" button is used to actually select the items.

For each line argument, the command adds all objects which intersect the line. This is useful for schematic drawings where paths (wires) are zero width. Selection using lines works best if the entered lines are manhattan. Non-manhattan lines may select more objects than intended.

For each box, the command adds all objects completely contained in the box to the list. Note that an object is considered contained if and only if its *bounding box* is *completely inside* the given box. The *select-objects* command is incremental; i.e. it may be called many times, each time adding to the selected set. All items selected are highlighted in the VEM highlight color.

select-terms ˆT [objs][pnts][lines][boxes]

This command selects all terminals (both actual and formal) whose implementations intersect the objects found by examining the argument list. The semantics for specifying the objects is identical to that described for *select-objects.* The items on the argument list will be replaced with the set of terminals found by examining the items. This command is useful for deleting formal terminals, specifying actual terminals for use by *edit-property* or *select-bag,* or creating new symbolic formal terminals using *promote.* As such, *it is extremely dangerous in* **Ptolemy**.

transform t [objs]

This command takes a selected set of objects built by selection commands and transforms them. The objects remain on the selected set. It is important to note that the actual objects in the database are not affected by this command. The transformation is associated with the object set not with the objects themselves.

The command takes (up to) three arguments: a set of objects to work on, a text rotation specification, and two points indicating a relative translation. The object set must be supplied. The rotation specification is a list of keywords separated by colons:

mx

Mirror around the Y axis.

my

Mirror around the X axis

90

Rotate counter-clockwise 90 degrees.

180

Rotate counter-clockwise 180 degrees.

270

Rotate counter-clockwise 270 degrees.

If both a rotation and a translation are given, the rotation specification should come first. If no rotation and translation are given, the command rotates the items 90 degrees counter-clockwise.

The *transform* command is incremental. This means it can be applied many times with each command having a relative effect on the current selected set. For example, invoking *transform* without arguments twice is the same as invoking it once and specifying the rotation string "180".

Once the command completes, the highlighted form of the objects will reflect the specified transformation. This can be used as a reference for further rotations or translations.

Translations are specified by two points. A relative translation is applied to the current transformation based on the vector formed by the two points. It is also possible to interactively specify the translation of a selected set. This is done by moving the cursor to a reference point and pressing, dragging, and releasing the right mouse button. While the right button is depressed, the selected objects will track the cursor motion. This can be used to drag items around interactively.

After completing a transformation with transform, the objects are not actually changed until a command that manipulates objects is invoked (e.g. move-objects, copy-objects, etc). See the *move-objects* and *copy-objects* commands below for more information.

unselect-objects u [objs] [pnts] [lines] [boxes]

The *unselect-objects* command is used to remove items from a previously created object argument (select-objects operation). Any number of points, lines, and boxes may be specified. The semantics for mapping these arguments to objects is the same as *select-objects.* For each point, zero or more items beneath the point that are part of the selected set are removed. For each line, all items in the selected set intersecting the line are removed from the set. For each box, all items completely contained in the box that are part of the selected set are removed from the set.

It is important to note that this command is intended to be used to unselect small sets of already selected objects. To unselect all items on the argument list, use control-W or control-U.

10.2.5. Property and Bag editing

The **Oct** Data Manager supports two kinds of annotation: properties and bags. Properties are named objects which may have an arbitrary value and can be attached to any other **Oct** object. Properties are used by pigi to store parameters. Bags are named objects which can contain any number of other **Oct** objects, and are not used by pigi. Bags are used to represent common collections of objects (instances for example) that can be accessed efficiently. There are a number of VEM commands used to create, edit, and delete properties and bags. However, since only one of these is useful in pigi, only one is documented here.

show-property ˆP [objs]

The *show-property* command produces a list of all the properties associated with the object under the cursor when the command was invoked (or all objects in the selected set). If there are many objects under the cursor, the command will present a dialog which lists each object. The user can select one by clicking the mouse in the check box next to the object name. The object will be highlighted. When the right object is found, the user can click "ok" to show the properties attached to the object. If the cursor is not over an object and nothing is in the selected set, the properties attached to the facet are shown.

The properties are shown in the form: xid: name (type) = value. The *xid* is the external identifier for the property and can be used in evaluated labels. The command also echos the type of the object each property is connected to.

10.2.6. Physical editing commands

The physical editing style in VEM is used by pigi for editing icons. The commands described below are available in addition to the common commands described in the previous section.

alter-geometry **a** **[box] [lines] or [points]**

This command replaces the box, path, or polygon under the cursor with the new specification supplied on the argument list. This can be used to "stretch" geometry or change their composition. For example, to make a box slightly larger, enter the slightly larger box onto the argument list, move the mouse over the old box, and invoke *alter-geometry.*

change-layer **l** **[objs] or [pnts][lines][boxes]"layer"**

The *change-layer* command detaches geometry from its current layer and attaches it to a different layer. The geometry can be specified either as an object set constructed by the selection commands, or directly by drawing points on them, drawing lines through them, or drawing boxes around them. Normally, the target layer is determined by looking at the geometry under the cursor when the command was invoked. However, if the last argument to the command is text, it is interpreted as the name of the target layer.

copy-objects **x** **[objs] {pnt pnt}**

The *copy-objects* command copies a set of objects from one place to another. The command takes an object argument that should contain a list of objects to be moved (this is built with *select-objects* and *unselect-objects).* The command assumes the object set has been transformed using the *transform* command or interactively dragged to a new location with the right mouse button. The command makes a copy of the objects where are transformed according to this translation. For example, to copy objects from one location to another, the user first selects the objects using *select-objects,* then interactively drags the objects using the right button (transformation), then invokes the *copy-objects* command to make a copy at the new location. Since the items remain selected, new copies can be made without reselecting the objects.

The optional second argument should be two points which specify the source and destination points of the copy. This alternative can be used if the object set is too large for interactive dragging or one wishes to copy objects from one facet into another. If the copy is from one facet to another facet, terminals will not be copied and the objects will be copied in a manner that preserves the position of the objects relative to the source point. The default key binding for this command is short for "xerox".

create-circle C **[line] [pnts] "layer"**

Since VEM does not have a circle argument type, a special circle drawing command has been added. For most types of geometry, the user should use *create-geometry.* Circles are specified in one of two ways. The first is a line followed by up to two points. The line specifies a filled circle with the first point being the center and the second its outer radius. If the first point is supplied, an arc is assumed with an angle formed by the second point of the line, the center point and the newly specified point. The angle is measured counter-clockwise. If the last point is supplied, it specifies the inner radius of the circle (otherwise the inner radius is zero). The second form takes two points and an optional point. It specifies a circle where the inner and outer radius are the same. If the last point is supplied, an arc with the same semantics as the first form is assumed. Finally, the layer of the circle is determined from the cursor position or by a final text argument specifying the layer directly.

create-geometry c **[pnts] [lines] [boxes] [text] "layer"**

The *create-geometry* command creates new geometry. It takes any number of points, lines, boxes, or text and a layer specification. A points argument creates a closed polygon. A line argument creates a multi-point path. Box arguments create boxes. Finally, text arguments create labels. When creating labels, the point set after the label is interpreted as the target points for the label. All the geometry is created on the same layer. This layer may be specified as the final text argument to the command or by invoking the command over an object attached to a layer. If the layer is ambiguous, the command will present a choice of layers in a dialog box. The palette command can be used to create a window which offers all possible layers for creating geometry.

create-instance <*> **[pnt] {"master:view name"}**

In most cases, the leaf cells designed with the physical editing style are not hierarchical. Instead, instances of the low-level cells are connected together using the symbolic editing style. However, those who would like to use VEM as a purely physical design editor require instances in physical cells. This command places instances in the physical editing style.

The instance is placed relative to the point supplied to the command. The master of the instance can be specified in two ways. In the first form, the user supplies a text argument which contains the cell, view, and instance names separated by spaces. The instance name is optional. The second form determines the master of the instance by looking at the instance under the cursor when the command is invoked. This instance can be in the same cell or in another cell. A common practice is to build a cell of primitives and use this cell as a menu for placing physical instances (see the *palette* command).

create-terminal <*> **["term name"]**

This command creates a new formal terminal named "term name". The implementation of the terminal is determined by constructing a list of all geometries under the spot where the command was invoked and choosing the smallest coincident boxes from this list. *This command is dangerous in pigi.*

delete-objects **D** **[objs] or [pnts] [boxes] "layer"**

The delete command removes objects from a cell. The command has two forms. The first takes an object argument constructed using selection commands and deletes all of the items in this set. The second takes any number of point, line, and box arguments and a layer specification. This form deletes all objects under the points, all objects which intersect the lines, and all objects completely contained in the boxes that are attached to the specified layer. The layer may also be specified by placing the cursor over some other object attached to a layer when the command is invoked. If no layer is specified, all of the geometry is deleted.

edit-label **E** **[pnt] { "LAYER" } or [objs]**

The *edit-label* command creates and edits labels. The command has two forms. The first form creates a new label at the specified point on the given layer. If the layer is not specified, it will be determined by looking at the object under the cursor. The second form edits labels selected using the *select* command.

Labels in Oct are represented as a box where the text is drawn entirely inside the box subject to justification and text height parameters. The edit-label command builds the box automatically by examining the text height and text itself. Thus, the user can control the justification, text height, and the label text. These parameters are set using the edit-label dialog box. This is a modeless dialog box that is posted when the user invokes the edit-label command.

The edit-label dialog box consists of three check-box areas for adjusting the label justification, and two type-in fields for adjusting the text height and the text itself. The justifications are computed in relation to the point the user specified when the label was first created. Thus, the horizontal justification specifies whether the point should be to the left, center, or right of the text. Similarly, the vertical justification specifies whether the point should be on the bottom, center, or top of the text. Finally, the line justification specifies how the lines should be justified within the text block when there is more than one line. The text height of the text is given in Oct units. Note that the X window system does not directly support fully scalable fonts. Thus, VEM uses a strategy where it will pick the closest font from a set of fonts that can be specified as a startup parameter (see the document "Customizing VEM" for details). Finally, the last type in field can be used to enter the text for the label. The label can have as many lines as necessary.

At the bottom of the dialog are four control buttons: *Ok, Apply, Dismiss,* and *Help.* The *Ok* button updates the value of the label and causes the dialog to close. The *Apply* button updates the value of the label (showing the effects in the graphics window) but does not close the dialog. This allows the user to adjust the label several times if necessary. The *Dismiss* button closes the dialog without updating the value of the label. Finally, the *Help* button displays some help about how to use the dialog.

move-objects **m** **[objs] {pnt pnt}**

The *move-objects* command moves a set of objects from one place to another. The command takes an object argument that should contain a list of objects to be moved (this is built with *select-objects and unselect-objects).* The command assumes the object set has been transformed using the *transform* command or interactively dragged to a new location using the right mouse button. The command moves the objects to a new location based on this transformation. For example, to move objects from one location to another, the user first selects the objects using *select-objects*, then interactively drags the objects using the right button (transformation), then invokes the *move-objects* command to actually move the items to the new location. The items remain selected for further moves if necessary.

The optional second argument should be two points which specify the source and destination points of the move. This alternative can be used if the object set is too large for interactive dragging. Unlike the *copy-objects* command, objects cannot be moved from one facet to another.

10.2.7. Symbolic editing commands

The symbolic editing commands in VEM allow the user to edit **Oct** symbolic views. Symbolic views are used to represent layout in a form suitable for compaction and simulation. Since they are not used by pigi, they are not documented here.

10.2.8. Schematic editing commands

The schematic editing style is an extension of symbolic. In addition to the general, selection, and options editing commands, the following commands are specific to the schematic editing style:

delete-objects **D** **[objs] or [pnts, lines, boxes] ["layer"]**

This command takes either an object list created with the *select-objects* and *unselect-objects* commands, or points, lines, and boxes with an optional layer-name. The resulting objects are deleted from the cell.

select-major-net **^N** **pnts, lines, or boxes ["net name"]**

This command finds the net associated with the object under the points, intersecting the lines, or inside the boxes and highlights all objects on that net. If no points, boxes, or lines are specified, the object under the cursor will be examined. Alternatively, if a net name is provided, objects on the named net are highlighted. This command can be used to check the connectivity of a symbolic cell. The command is incremental (i.e. multiple nets can be selected).

move-objects **m** **[objs] {pnt pnt}**

The *move-objects* command moves a set of objects from one place to another. The command takes a set of objects (specified using select-objects) and a transformation. This transformation can be specified in two ways: as two points specifying a relative translation or using the transform command. The transform command allows general mirroring and rotation to be specified in addition to a translation. It is also possible to specify the translation of an object set by interactively moving the set using the right mouse button (see the transform command). After a move operation, the objects remain on the argument list for further move or copy operations.

For example, a set of objects can be moved using the following sequence: select the objects using the select-objects command. Move the cursor over a reference point. Depress and hold the right mouse button and drag the items to a new location. Use the transform command to rotate or mirror the object set. Once everything is correct, invoke the move-objects command to actually move the objects.

The connectivity of the items moved by this command is not changed. This means an instance moved using this command also causes the segments attached to its terminals to move as well. Moving objects between facets is not supported.

copy-objects **x** **(See Below)**

The *copy-objects* command copies a set of objects from one place to another. This command has the same form as *move-objects* (see above) except that the objects are copied not moved. Like *move-objects,* the objects copied remain on the argument list for further move and copy operations. However, unlike *move-objects,* the connections to the objects in the selected set are not copied. Instead, the connectivity between the selected items is copied along with the objects.

create **c** **(See Below)**

The *create* command allows the user to add new objects to a schematic cell. Different arguments given to *create* will produce different objects.

Formal Terminals - "terminal-name [type] [direction]" : create

When *create* is given a single argument string, the name of a new formal terminal, a formal terminal is created. The implementation of the formal terminal is taken to be the actual terminal currently under the mouse (note: a connector terminal can also be used for this purpose). Since terminals in schematic may be quite small, this routine will try to find nearby terminals if it doesn't find a terminal directly beneath the cursor.

Formal terminal names must be unique within the cell. If a formal terminal of the given name already exists, VEM will display a dialog box asking whether or not you wish to replace the old terminal.

Two optional pieces of annotation can be placed on the terminal: type and direction. The type can be one of SIGNAL, CLOCK, GROUND, SUPPLY, or TRISTATE. If not provided, SIGNAL will be assumed. The direction can be one of IN, OUT, or INOUT. If not specified, INOUT will be assumed. If either of these values are not provided in the terminal name specification, VEM will post a dialog box containing fields for entering both the terminal type and direction. Pressing the left mouse button in the value area for these fields will cause a menu of the possible choices to appear. New values can be selected by releasing the mouse button over the desired value. Once appropriate values are selected, activating the *Ok* button at the bottom of the dialog will save the annotations. Activating the *Dismiss* option will leave the annotations unspecified. These annotations can be edited later using the edit-property command.

Instances - points [[master:view] **] : create**

If the arguments to *create* are a number of points followed by an optional string, instances will be created with their origins at those points. If the name of the master is not specified textually, it will be inferred from the instance under the mouse.

The string argument has two parts: the instance specification and name. Both of these fields are optional. Specifying a null string is considered to be the same as no string at all. The instance specification is a master-view pair, such as "amp:schematic". If this field is left out, the master is inferred from the instance under the mouse. Otherwise, an attempt is made to locate the instance by the master-view pair. If the name field is given, the instance will be given the specified name.

NOTES: Newly created instances whose actual terminals intersect actual terminals of other instances will be automatically connected. In this case, no path is required between the terminals. Rotated and mirrored forms of instances can be created by instantiating a new instance and using the transform and move-objects or transform and copy-objects commands.

Paths - lines : create

This command creates new segments for connecting together instance actual terminals. A new series of segments will be created on a predefined layer (WIRING). Connector instances will be placed automatically at all jog points. The width of the new path is always zero.

Normally, the schematic editing style has a feature turned on called "gravity". When you draw segments with gravity turned on, VEM will try to connect the segments to a nearby terminal if you miss a terminal by a small amount. This is useful when editing a large cell.

edit-label E [pnt] { "LAYER" } or [objs]

The *edit-label* command creates and edits labels. The command has two forms. The first form creates a new label at the specified point on the given layer. If the layer is not specified, it will be determined by looking at the object under the cursor. The second form edits labels selected using the *select* command.

Labels in Oct are represented as a box where the text is drawn entirely inside the box subject to justification and text height parameters. The edit-label command builds the box automatically by examining the text height and text itself. Thus, the user can control the justification, text height, and the label text. These parameters are set using the edit-label dialog box. This is a modeless dialog box that is posted when the user invokes the edit-label command.

The edit-label dialog box consists of three check-box areas for adjusting the label justification, and two type-in fields for adjusting the text height and the text itself. The justifications are computed in relation to the point the user specified when the label was first created. Thus, the horizontal justification specifies whether the point should be to the left, center, or right of the text. Similarly, the vertical justification specifies whether the point should be on the bottom, center, or top of the text. Finally, the line justification specifies how the lines should be justified within the text block when there is more than one line. The text height of the text is given in Oct units. Note that the X window system does not directly support fully scalable fonts. Thus, VEM uses a strategy where it will pick the closest font from a set of fonts that can be specified as a startup parameter (see the document "Customizing VEM" for details). Finally, the last type in field can be used to enter the text for the label. The label can have as many lines as necessary.

At the bottom of the dialog are four control buttons: *Ok, Apply, Dismiss,* and *Help.* The *Ok* button updates the value of the label and causes the dialog to close. The *Apply* button updates the value of the label (showing the effects in the graphics window) but does not close the dialog. This allows the user to adjust the label several times if necessary. The *Dismiss* button closes the dialog without updating the value of the label. Finally, the *Help* button displays some help about how to use the dialog.

10.2.9. Remote application commands

The following commands apply only if a remote application, like pigiRpc, is running.

kill-application **<*>** **No Arguments**

> The *kill-application* command kills the remote application which has control of the window where the command was invoked. This can be used to terminate runaway remote applications. Note it has no standard menu or key bindings; it is a type-in command only. This command may not work on all machines.

rpc-any **r** **"host pathname"**

> This command starts up a remote application which is not on the standard list of applications in the VEM menu. "host" specifies the network location for the application and "path" specifies the path to the executable on "host". If the host specification is omitted, the local machine is assumed.

rpc-reset **ˆR** **No Arguments**

> If an application terminates abnormally, VEM may not recognize that the window no longer has an application running in it. This command forces VEM to reset the state of the window so that new applications can be run in it.

10.2.10. Customization

VEM is a highly customizable editor. Nearly all of the colors, font styles, and fill patterns it uses can be configured to a user's taste. See "Customizing VEM" for more details.

Furthermore, VEM is meant to be an extensible program. New commands can be written that are either tightly bound to the editor or that run in a separate process space. There are documents describing both tightly bound and remotely bound extensions to VEM. See the complete octtools distribution documentation for details.

10.2.11. Bugs

Opening a facet that is inconsistent (either out of date or one with conflicting terminals) isn't handled very gracefully. You can open an inconsistent facet by opening it again.

Bounding boxes may not be drawn if there is no geometry in the cell.

The set path width command doesn't work if you use a palette to specify the layer.

Editing the master of an instance displayed in some other window sometimes causes the instance to disappear. Re-reading the cell with the instance gets around the problem.

Send bug reports to vem@eros.Berkeley.EDU

10.3. Customizing VEM

Watch this space. More coming later.