Ptolemy: A Mixed-Paradigm Simulation/Prototyping Platform in C++

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

September 6, 1991
Dept. of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

Ptolemy is a flexible and extensible platform for simulations, rapid prototyping, and other software systems. It is our third generation software environment, building on our experience with Blosim, a signal processing simulation system, and Gabriel, a prototyping environment for real-time signal processing. Unlike its predecessors, Ptolemy does not enforce a single simulation or execution model; it supports mixed hardware and software designs, mixed-mode system descriptions, and mixed prototyping methodologies.

The objectives, design, current status, and future directions of the Ptolemy project are summarized in this paper. The use of certain features of the C++ language to make it easier to design a flexible, extensible system is discussed. We also describe our experiences in developing a 150K line (and growing) system in C++, discussing what worked well, what didn't and lessons learned.

1. INTRODUCTION

Ptolemy is a flexible and extensible platform for simulation, rapid prototyping, and related design environments. Ptolemy has few assumptions built into it, and thus can be used with the same generality as building a system from scratch. Ptolemy provides, however, a large set of building blocks that can be used to reduce the effort of building new environments. Most importantly, Ptolemy defines a set of internal object-oriented interfaces, so that heterogeneous environments built in the Ptolemaic framework can later be easily merged as necessary.

Ptolemy was devised in an attempt to overcome some of the shortcomings of its predecessors, Gabriel and Blosim, which were restricted to particular assumptions about how systems were modelled. To preserve the advantages of the old systems while removing their restrictions, it was clear that we required object-oriented programming methodology.

The systematic application of object oriented programming to simulation problems began with the Simula programming language, which is one of the two ancestors of the C++ programming language (the other being, of course, C). In fact, C++, or ''C with classes'', as the first version of the language was called, was first used in discrete-event simulation [Str86].

While a language such as Smalltalk or an object-oriented Lisp could have been used, we were interested in computational efficiency and also in the clean integration of signal processing algorithms and other code written in C, so these considerations narrowed down the choice of languages to C++ and Objective-C. The wider availability of compilers and tools for C++ at the time the project was beginning (1989) helped to sway our final choice.

1.1. Organization of Ptolemy

The basic module in Ptolemy is the *block*. The entity that determines the order of execution of blocks is the *scheduler* (at either compile time or run time or some combination). The combination of a scheduler and a set of blocks that conform to the behavior expected by this scheduler is called a *domain*. Different models of computation (time-driven, event-driven, etc) can be built on top of Ptolemy by simply substituting different domains. Two or more simulation environments built on Ptolemy can be combined into a single environment, thus enabling *heterogeneous* simulations of large systems that combine different schedulers and computational models. New domains are easily added to Ptolemy, including domains that do not conform to the block/scheduler model. In addition, new blocks and domains may be added to a running system by means of incremental linking.

Previous simulation systems have a fundamental model of computation and scheduling built in at the foundation; Ptolemy does not. This enables Ptolemy to mix different models of computation, virtually without limitation, within the same system. While Ptolemy was first conceived for simulations, it is a useful platform for constructing many types of software systems, examples including multiprocessor code generators, telecommunications feature software, hardware-software codesign, scientific computations, etc. Ptolemy is named after a famous astronomer because of the extensive use of cosmological metaphors in its basic structure.

A key objective of Ptolemy is to allow many different computational models to coexist in the same system. This is quite distinct from previous systems, where a single inflexible model of computation is typically supported (event-driven simulation, message queue paradigm, time-driven simulation, etc.) Ptolemy defines the concept of a *domain.*

Since the standard graphical user interface of Ptolemy displays block diagrams, it is convenient in Ptolemy to divide a computation into a set of *blocks*. A block is a module of code, which is *invoked* at run-time, consumes data present at its inputs and generates data on its outputs, and that is allowed to run to completion once it is invoked. Typically "completion" means deadlock, in the sense that all input data is consumed. This doesn’t limit the generality of Ptolemy since literally anything can be embedded within a block (and in the limit the whole system could be a single block!). But normally, we try to divide an application development into blocks that are small and manageable and that communicate with one another through messages. This allows the blocks to be developed and tested independently from one another. Ptolemy also encourages developers to document blocks and store them in standard libraries; it accomplishes this by providing a preprocessor that produces not only C++ code for the block but also a manual entry describing its use. Thus, blocks in Ptolemy become modular, reusable software components.

A *domain* in Ptolemy consists of a set of blocks that obey a common *computational model*. Within a block-diagram representation, there are two aspects to the computational model:

- the semantics of how a block interacts with other blocks, and
- the operational semantics of the order of invocation of the blocks.

1.2. Models of Computation

Some examples of general-purpose domains that are currently available within Ptolemy or are currently being designed include:

- *Dynamic dataflow (DDF)* is a general model for signal processing that includes asynchronous operations, and is the model embodied in the predecessor system Blosim [Mes84a,Mes84b]. In DDF, blocks consume generate data sporadically in data-dependent fashion, and the order of block invocation is in accordance with data precedence relationships. The DDF does not attempt to model the relative timing relationship of block invocations.
- *Synchronous dataflow (SDF)* [Lee87a] is an appropriate model for signal processing systems with rationally-related sampling rates throughout, and is the model used exclusively in the predecessor system Gabriel [Lee89a]. In SDF, blocks consume and generate a static and known number of data tokens on each invocation. Advantages of SDF are ease of programming (since the availability of data tokens is static and doesn't need to be checked), a greater degree of setup-time syntax checking (since sample-rate inconsistencies are easily detected by the system), and run-time efficiency (since the ordering of block invocation is statically determined at setup-time rather than dynamically at run-time).
- *Discrete event (DE)* is a model in which only changes in system state (called events) are modeled. This is an asynchronous model like DDF, but unlike DDF incorporates the concept of global time in the system, and orders block invocations properly in time. A completely general simulation system could be developed in the DE domain, at the expense of run-time efficiency and ease and naturalness of programming for many applications like signal processing.
- *Message queue (MQ)* is a model similar to DDF but with many more capabilities for dynamically creating and destroying blocks. The MQ domain is another experimental domain under development targeted at software control applications, such as telephone switching call-processing software.

In addition to these general-purpose domains, the capability to create domains out of previously-existing simulation systems has been demonstrated with the *Capsim* domain (incorporating the Capsim signal processing system [Fab] that is based on Blosim) and the *Thor* domain (incorporating the Thor hardware timing simulator [Thor]).

1.3. Design Goals

Ptolemy is motivated by the increasingly important role of high-level system design, the increasing proliferation of simulation platforms and computational models, and the increasing need to combine these computational models. Some of the goals of Ptolemy include:

- Do not limit the generality by building unnecessary assumptions or limitations into the basic system. Rather, provide a set of building blocks that can be profitably used to more effortlessly construct future systems.
- Specify each subsystem of an application using a domain most natural for that subsystem, and yet seamlessly combined with other subsystems to form a heterogeneous whole.
- Accept without modification subsystem descriptions from previous simulation platforms, where appropriate, and integrate them seamlessly into a Ptolemy simulation.
- Allow future domains to be added without any modifications required to Ptolemy itself or previous models and schedulers.

- Encourage users to build reusable software components by defining standard interfaces among components, standard documentation procedures, and version management.

2. EXAMPLE OF A MIXED-DOMAIN SIMULATION

To illustrate the application of Ptolemy consider the simulation of a broadband packet network. In figure 1a, a high-level view of a terminal for a broadband packet network is shown, emphasizing the partitioning of functionality into signaling and control (call establishment, flow control, multimedia synchronization, etc.), and packet network. Typically the speed of the control components is sufficiently high that their detailed timing can be neglected, in which case a domain such as MQ is suitable for their modeling. In contrast, the propagation delays in network links and in the control messages between control nodes are often critical to synchronization issues, and should be modeled with detailed timing, such as in the DE domain. In figure 1b, the network transport of one specific service — packet speech — is illustrated. This transport divides into two pieces, the signal processing (compression, silence detection) that is best modeled with a time-driven synchronous sampling rate (like the SDF domain), while the network (packet assembly/disassembly, switching and queueing) is best modeled by only changes in system state (like the DE domain).

All these distinct system components interact in important ways that need to be studied by simulation. Especially in multimedia services (combinations of video, voice, data) there is interaction between the control and transport components. In the speech portion, statistical network delays and packet loss interacts with the signal processing design. Ptolemy allows these interaction issues to be studied, while maintaining a natural representation of each system component.

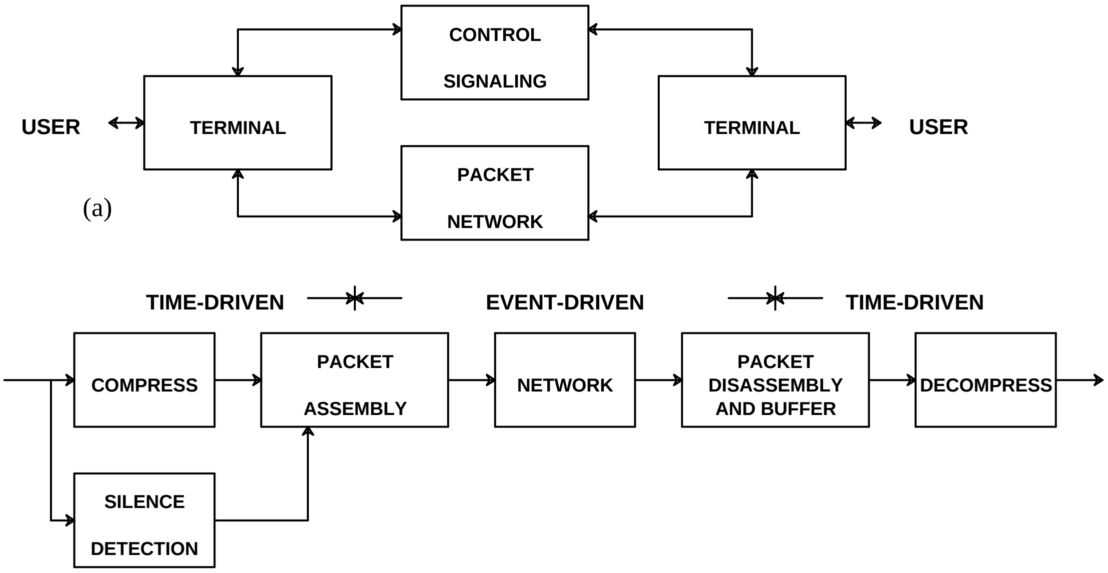


Figure 1. A packet speech system simulation requires the combination of signal processing (compression, silence detection) and queueing (packet assembly/disassembly and packet transport).

3. ACHIEVING HETEROGENEITY — POLYMORPHISM

3.1. Messages in Ptolemy

We can illustrate the use of polymorphism in Ptolemy first by looking at the form of data or messages that pass between blocks.

Because of the common base class, all messages can be dealt with as `Particle`s regardless of what type of information or data they contain. Ptolemy is simply not cognizant of all the different types of `Particle`s that inherit base `class Particle`, and hence new types of `Particle`s can be added to Ptolemy without affecting any of the Ptolemy code.

Shown in figure 2 is a representative set of methods for `Particle` (the list of methods in all our examples is actually much longer) including:

- *initialize()* sets a default value for a `Particle`, such as zero for a `FloatSample`. (All objects in Ptolemy have an *initialize()* method, used to begin or restart a simulation.)
- *operator float()* converts a `Particle` to a `float` value, if that makes sense (for example if it is an `IntSample` or `FloatSample`). If it does not make sense, for example if the `Particle` is a `SpeechPacket`, then a value of zero is returned with a proper error message. In addition, an error condition is asserted that will cause the scheduler to halt; eventually, exceptions will be used to accomplish this more cleanly.
- *operator =()* is the assignment operator, which allows one `Particle` to be copied to another identical type of `Particle`.
- *clone()* creates a new `Particle` of the same type. Initially Ptolemy maintains a list of prototype `Particle`s of all known types, and new `Particle`s can easily be created by this prototype using this method.
- *print()* turns a `Particle` into a character string in some appropriate format, useful for printing it out to the user interface.

All these methods are virtual, which permits, for example, automatic type coercion when, for example, a block that produces integers is connected to a block that expects complex values. Examples of types that can be derived from base class `Particle` are also shown in figure 2 (the inheritance hierarchy is somewhat simplified).

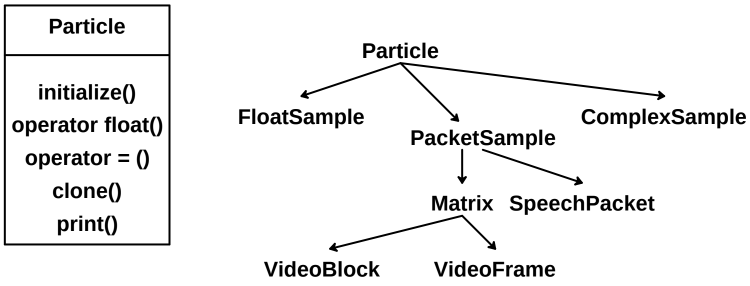


Figure 2. All messages in Ptolemy are derived from a base `class Particle`. Some representative methods of this base `class` are shown, as well as some typical types of `Particle`s that can be derived from `Particle`.

During execution of complex simulations, large numbers of `Particle`s are (at least conceptually) created and destroyed. To make Ptolemy as efficient as possible, pools are maintained of fixed-size objects such as `IntSample` and `ComplexSample`; `PacketSample` uses a reference-count mechanism so that pointers, rather than whole objects, are copied as particles move around the system.

3.2. Functional Blocks in Ptolemy

The basic module of computation in Ptolemy is the `Block`, illustrated in figure 3. An atomic block, called a `Star`, is a type of object that typically performs some unit of computation within the simulation, and communicates with the outside world through a stream of `Particle`s passing through its `PortHole`s, always of the same type. For the most part, `Particle`s either leave or enter a given `PortHole`, but not both; however, bidirectional ports are possible and are used in the Thor and MQ domains.

There are many different types of `Star`s, but they all interface to Ptolemy and to each other in a standardized way through polymorphism. The only constraint is the basic interface that must be provided, and certain standard methods that must be provided. A sampling of methods is shown in figure 3:

- The four stages of a simulation are *initialize()* (allocate memory for data structures, etc.), *start()* (set internal parameters and state variables, etc. at the start of a new simulation), *go()* (perform one step of the simulation), and *wrapup()* (collecting statistics etc.).
- *clone()* makes a copy of this type of `Block`, for example to create another instance.
- *mySched()* returns the `Scheduler` for this `Block` (more on `Scheduler`s later).
- *domain()* returns the name of the domain within which this `Block` is intended to operate.

Also shown in figure 3 is an inheritance diagram illustrating how customized `PortHole`s can be created for different domains. In particular, it is shown how input and output `PortHole`s customized to the SDF domain are inherited from the base `class PortHole`.

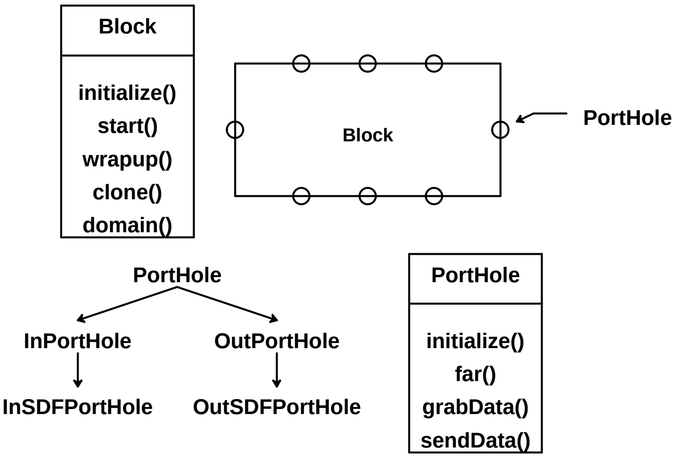


Figure 3. The `class Block` models certain behavior common to all computational units in Ptolemy. `PortHole`s are the interface to other `Block`s, where each `PortHole` carries a stream of `Particle`s. Derived types of `Block`s are shown later in figure 5. Also illustrated are some of the typical methods in `class Block`.

3.3. Schedulers

The function of the `Scheduler` is to determine the order in which `Block`s are executed. For example, the basic difference between the time-driven and event-driven styles of simulation is the way in which the order of the `Block` invocation is determined. Some of the methods of `class Scheduler` are shown in figure 4, including *setup()* and *run()* which perform the compile-time and run-time scheduling functions respectively, and *setStopTime()* which tells the `Scheduler` before a *run()* the period of time to allow to elapse before stopping. Different `Scheduler`s corresponding to different domains simply substitute different implementations of these methods. For example, in the SDF domain *setup()* establishes a static order of invocation of the `Block`s, and *run()* then simply executes that invocation order repetitively until the criterion of *setStopTime()* is established. In the DE domain, most of the activity is in method *run(),* which manages an event queue for the dynamic ordering of `Block` invocation.

3.4. Parameters and States

Ptolemy provides a group of classes derived from class `State` for use as parameters and state variables for functional blocks. Derived classes provide states of various types, for example, `IntState`, `FloatArrayState`, `StringState`. Each state has two values: the first, provided by the base class, is a character string that holds an expression used to initialize the state; the second, whose type matches that of the state, is the run-time value of the state. Each type of state provides a recursive-descent parser that evaluates the initializing expression when execution begins; these expressions look much like arithmetic expressions in C or C++. A state belonging to a galaxy may have its name used in expressions for states belonging to blocks within that galaxy; this permits galaxies to have parameters that affect the operation of the blocks they contain.

States have attributes that indicate whether their values can change at runtime or not (as well as other properties of the state); these attributes may be changed by derivation. For example, class `SDFFIR`, an finite impulse response filter star, has a fixed `FloatArrayState` specifying the filter taps; class `SDFLMS`, an adaptive filter using the LMS algorithm, is derived from `SDFFIR`; since the filter is adaptive the filter taps now change.

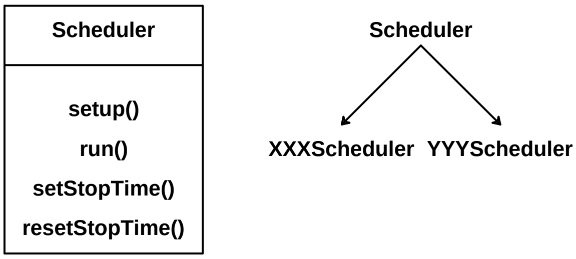


Figure 4. An example of inheritance and some selected methods for a `Scheduler`.

4. DOMAINS and WORMHOLES

The key concepts in Ptolemy that support heterogeneous simulations are the domain and the `Wormhole`. Thus far the basic module of simulation, the `Block`, has been described, but in fact there are several distinct types of `Block`s. A simplified inheritance diagram for `Block`s is shown in figure 5.

At the top level, `class Block` contains instance data and methods associated with the `PortHole`s of the `Block`, as well as its name and other relevant data. In addition, base `class Runnable` encompasses the instance data necessary for those blocks that have an internal `Scheduler`, such as a pointer to the `Scheduler` itself. The `Galaxy` and the `Star` are the two basic types of `Block`. The `Galaxy` is a `Block` that internally contains a collection of interconnected `Block`s; thus, it is provided solely for the purpose of a hierarchical description of `Block`s. All the `Block`s in a `Galaxy` are assumed to be homogeneous with respect to the `Scheduler` (for example, they all use a time-driven or an event-driven `Scheduler`). `Star`s on the other hand are considered atomic or indivisible. This is illustrated in figure 6a, where a `Galaxy` contains `Star`s and other `Galaxy`s.

At the first stage of description of the cosmology of a particular simulation, the `Galaxy`s are *flattened*, and the hierarchical description is converted into a non-hierarchical description of

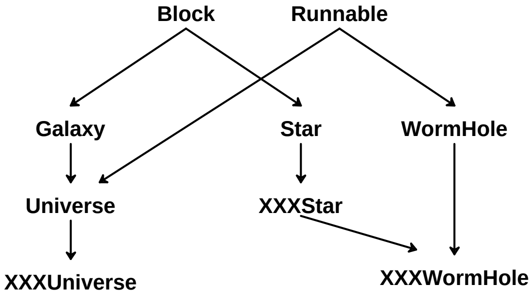
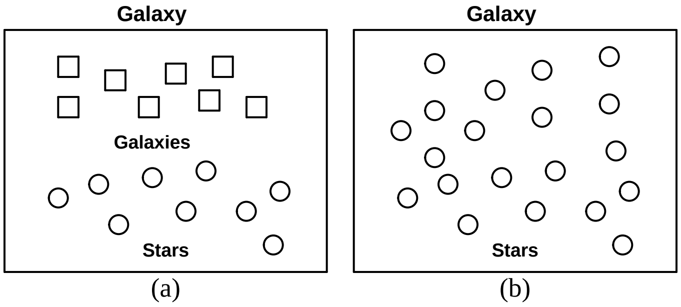


Figure 5. An inheritance diagram for `Blocks`, and some classes derived from `Block`. The `class Runnable` includes methods required for blocks that have an internal `Scheduler`. Shown are the classes needed to execute a fictitious domain "XXX".



(a) (b)

Figure 6. A `Galaxy` is a `Block` that internally contains a collection of `Stars` and `Galaxys`. (a) Before flattening, and (b) after flattening.

interconnected `Stars`. This is illustrated in figure 6b. The basic difference between a `Galaxy` and a `Star`, then, is that schedulers can ‘‘see’’ the internal structure of `Galaxy` objects but not of `Star` objects. When the simulation actually commences, a simulation consists of only `Star`s. As we will see, this does *not* mean that it is not hierarchical, but only that the simple form of hierarchy embodied in the `Galaxy` has been removed by flattening.

Within a given domain, all the `Star`s behave in a fashion appropriate for that domain, and there is a single `Scheduler` that assumes that behavior. For example, in figure 5 we show the inheritance diagram for a single domain, a fictitious domain XXX (where "XXX" might be "SDF", or "DE", etc.). In this domain, there is an `XXXScheduler` and an `XXXStar`, where `XXXStar` has behaves in a manner expected by the `XXXScheduler`.

The outermost `Block`, which contains the entire simulation, is known as the `Universe`. The `Universe` consists internally of a `Galaxy` and an associated `Scheduler`, and hence it inherits the characteristics of both a `Galaxy` and a `Runnable`. In our simplified cosmology of figure 5 there is only one type of `Universe`, the `XXXUniverse`. The internal structure of the `XXXUniverse` is illustrated in figure 7.

Different domains can be realized by implementing different `Universe`s, each containing an appropriate `Scheduler`. This does not by itself, however, allow multiple domains to coexist. Multiple domains interact by means of the special object `Wormhole`. The `Wormhole` is a special type of `Star` that behaves externally like any other `Star`, and thus can be freely mixed with other `Star`s of the same type. Internally, however, the `Wormhole` is special — it is like a `Universe` in that it contains its own `Scheduler`, which is always a different `Scheduler` from the one invoking the `Wormhole` externally. A `Wormhole` also contains an internal `Galaxy`, and in this respect is similar to a `Universe`. Roughly speaking, we can think of a `Wormhole` as a special `Universe` that unlike a normal `Universe` has `PortHoles`. Continuing this analogy, a `Wormhole` is like a `Star` that internally contains another `Universe`!

An example that illustrates the use of a `Wormhole` to mix two fictitious domains, "XXX" and "YYY" is shown in figure 8. For this example, a single `XXXWormhole` of YYY domain resides in an `XXXUniverse` with other `XXXStars`, where it appears and behaves just as if it were an `XXXStar`. Internal to the `XXXWormhole`, the `YYYStar`s are invoked by a `YYYScheduler`.

The way in which polymorphism makes `Star`s and `Wormhole`s look identical externally is illustrated in figure 9 (which is highly simplified). The *start()* and *go()* methods of these

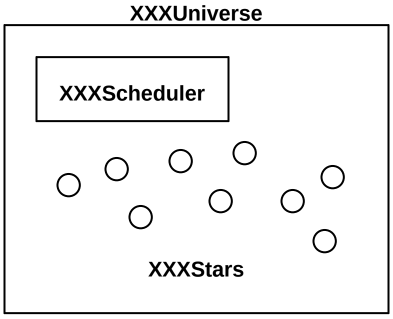


Figure 7. The internal structure of an `XXXUniverse`.

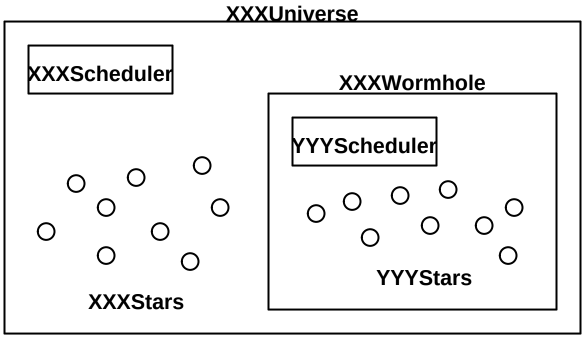


Figure 8. An example of a `Wormhole`, namely an `XXXWormhole` resides in an `XXXUniverse` with a collection of other `XXXStars`.

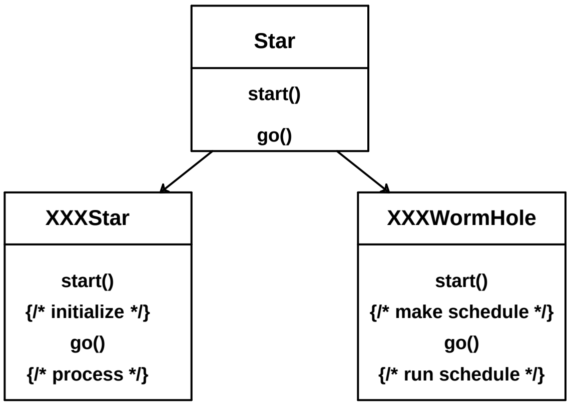


Figure 9. How polymorphism makes `Wormholes` and `Stars` look identical externally.

two classes perform quite different functions. For a `Star`, they perform some direct simulation function, and for a `Wormhole` they call `Scheduler` compile-time and run-time methods for the `Blocks` within the `Wormhole`. The externally visible effect is the same: a `Wormhole` looks like a `Star` from the outside, a principle that must be remembered in designing a `Wormhole` for a specific domain.

To summarize, a `Wormhole` embeds a heterogeneous domain into a Ptolemy simulation, where the external domain is not even aware that one or more `Stars` actually contain an entire foreign domain. As shown in figure 5, an `XXXWormhole` assumes the properties of both an `XXXStar` (which is how it appears externally), and also a `Wormhole` inheriting a `Runnable` (because unlike vanilla `Stars` it also contains an internal `Scheduler`).

4.1. Event Horizon and Universal Event Horizon

The `PortHoles` of a `Wormhole` are special in that they perform a conversion function between different `Schedulers`. Because of this crucial interface, the boundary between the input and output of a `Wormhole` is implemented by a special `class EventHorizon`.

There are some domain-specific operations to do in the `EventHorizon`. We could customize each domain interface by defining a `EventHorizon` class for each pair of domains. This approach is not attractive since if we have N domains, we would need N^2 types of `EventHorizon`. 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.

To avoid these difficulties, we have designed a *universal* `EventHorizon`, where each domain has only a pair of `EventHorizon`s. Now, we have only $2N$ `EventHorizon`s for N domains, and each domain is totally independent of the others. All `Particle`s passing into and out of a `Wormhole` actually pass through a pair of `EventHorizons` of which one is from the outside domain and the other is from the inside domain. This mechanism depends on the assumption that the tasks of domain interface can be cleanly divided into some tasks of the outside domain and the other of the inside domain, where the boundary is called the universal `EventHorizon`. We consider this approach experimental since it has been tried on a relatively few domains.

The idea of the universal `EventHorizon` is illustrated in figure 10. An input `PortHole` of a `XXXWormhole` of YYY domain residing in an `XXXUniverse` internally consists of two `EventHorizon`s, an `XXXtoUniversal` and a `YYYfromUniversal`. These two objects together perform the tasks of domain interface while delivering the `Particles` from the outside to the inside domain. Similarly, an output `PortHole` of the Wormhole consists of a `YYYtoUniversal` and an `XXXfromUniversal`. Also shown is an inheritance diagram, where we see that ultimately these classes are `PortHole`s.

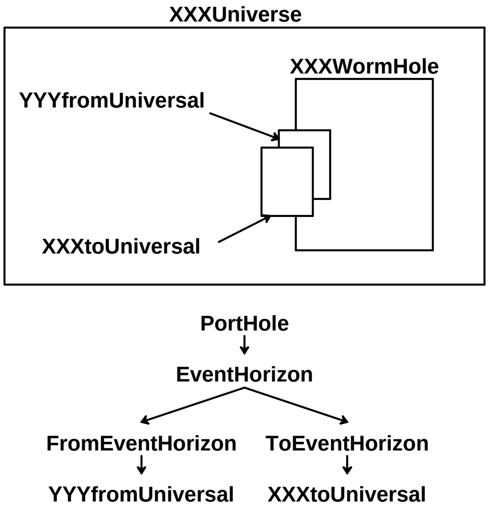


Figure 10. Illustration of the universal `EventHorizon`, where particles to and from a `XXXWormhole` of YYY domain pass through a pair of `EventHorizons`.

5. PUTTING THE PIECES TOGETHER

Ptolemy makes repeated use of several simple concepts to keep track of what functional blocks exist, what domains are supported, and to provide mechanisms for building simulations.

The first major concept is that of the prototype instance and cloning. Given a list (or other data structure) of objects of different types but a common base class, each indexed by a name (a character string), it is possible to create a new instance of a class by specifying its name, searching the list for a match, and, when the match is found, calling the virtual `clone()` function on the matching object. This technique is used in Ptolemy to allow for the dynamic creation of many types of objects.

However, we want to avoid introducing an explicit ‘‘master list’’ anywhere in the system. We would prefer to simply link together object files that contain prototype instances of all the classes we wish to support, and have the linking process somehow create the master list. To do this, we introduce classes of dummy objects whose constructors serve the function of building these lists.

The `KnownBlock` class manages the collection of blocks known to the system. A separate list of blocks is maintained for each domain. By convention, stars in a particular domain have a class name that is formed by concatenating the domain name with the ‘‘user interface’’ name; for example, the Fork star in the SDF domain is of class `SDFFork`. It is not necessarily true that each object on the list is of a different class, however; all galaxies created by means of the user interface are of class `InterpGalaxy` (described later).

As hinted above, we construct objects of class `KnownBlock` for their side effects only. Constructors have a very desirable property: when a global or ‘‘file-static’’ object is created, its constructor is called before execution of the program begins. We exploit this feature to create the known list of blocks, without requiring that any table appears anywhere in the programming giving a list of exactly which blocks are in the system.

It works as follows: class `KnownBlock` has a constructor that takes two arguments: a reference to a `Block`, and a `const char*`; the latter argument specifies the ‘‘user interface’’ name of the block. The constructor adds the block to the known list for its domain (there is a separate list for each domain), indexed by the supplied name. The static member function `KnownBlock::clone`, which takes a single argument, a const char*, creates a new Block of the given type from the list corresponding to the current domain. All of `KnownBlock`’s member functions are static except for the constructor, which is called only for its side effects. The effect of constructing a KnownBlock object is to register a block on the known list. We use this feature to advantage when we define new stars (or other types of blocks): if we say

```
static SDFFIR proto;
static KnownBlock entry(proto,"FIR");
```

in the file that defines class SDFFIR (e.g. a finite impulse response filter in the SDF domain), and also implement the member function

```
Block* SDFFIR::clone() const { return new SDFFIR;}
```

that’s all we need to have this scheme work. To make it easier, we provide a preprocessor to write stars; the preprocessor allows member functions to be written in C++ while inserting the ‘‘boilerplate’’ code for supporting the prototype instance, the clone function, and certain standard initializations for `PortHole`s and `State`s.

This scheme suggests a very natural approach to extending the system with incremental linking: if we can arrange to load in new object files and call the static constructors in the file, we have then extended the system and we know about a new functional block, and there is no strong distinction between a block added this way and one that was compiled in (actually there is a distinction, in that we flag blocks that are incrementally loaded and permit them to be redefined, but we cannot at present redefine compiled-in stars (the linker would complain about duplicate symbols, though in theory we could modify the linker). This scheme is similar to the system described in [Dor90], though it was developed independently.

The exact code required to call the global constructors in an object file is implementation-specific, but the general procedure is almost identical in, for example, *g++* and compilers derived from AT&T's *cfront*. There are also differences depending on the format of the object files: BSD-style or COFF. In any case, the general approach is to use the incremental linking facilities of the Unix linker on all these systems, and to somehow call the constructors for the global objects after the code is loaded in. At present, we support incremental linking with *g++* on all the platforms that Ptolemy runs on: the Sun-3, Sun-4 (Sparc), Vax, and DecStation (MIPS). The DecStation uses COFF so the details of the incremental link are different. We also support incremental linking under Sun C++ (based on the 2.1 version of *cfront*).

To allow the user to build and run a `Universe` out of system-supplied and user-written blocks, a dynamic galaxy class, `InterpGalaxy`, is provided. This class, which is derived from `Galaxy`, has the ability to modify itself by adding new stars, adding new connections between stars, adding aliases (mappings between portholes belonging to the galaxy and those belonging to interior stars), adding states, setting values for states of member stars, or removing stars and connections. Finally, it can add itself to the known list maintained by KnownBlock, and its clone function is capable of making a duplicate galaxy. All of its arguments can be specified solely with character strings: add a block of type ‘‘FloatRamp’’ and name it ‘‘source’’, etc. User interfaces use objects of this class, and of the class `InterpUniverse` (derived from `InterpGalaxy` and `Runnable`), to create and execute a ‘‘world’’.

It is undesirable to require the user interface to know any of the details of a specific domain. For that reason, there is a class named `Domain` that manages all the information about a domain: how to create a scheduler, a wormhole, or `EventHorizon`s for that domain, for example. To implement a new domain, a new class, for example, `SDFDomain`, is created, and a single prototype instance is constructed. Again, as for blocks, this prototype instance is added to a master list, and the system now knows how to create SDF universes, by searching the domain object list for the ‘‘SDF’’ object and asking that object.

Ptolemy provides two very different user interfaces, a text-based interpreter that accepts a Lisp-like language, and a graphical interface based on VEM, the graphic editor that is part of Berkeley's Octtools CAD system [Har89]. The interpreter is implemented as a class and is clean enough to embed in other programs. The graphic interface, on the other hand, was ported from the Gabriel system [Lee89] and is quite a bit harder to maintain. Both user interfaces work, for the most part, by translating user commands into requests to objects of class `InterpGalaxy` and `InterpUniverse` to build and execute simulations.

6. COMMENTS ON OUR EXPERIENCE WITH C++

In this section we discuss our experience with the C++ language and how we dealt with several issues that are controversial in the C++ community (as judged primarily be discussion in the Usenet newsgroup `comp.lang.c++` and in papers such as [Car91]).

When the Ptolemy project began, none of us had any experience with an object-oriented programming language. As several of us had extensive experience with the management of medium- to large-scale software systems, we had a good grasp of the importance of data abstraction and information hiding, but nethertheless, we needed to learn by doing.

As Ptolemy was not the first system of its kind, we did not have difficulty determining what some of the key objects were; while there have been changes in the inheritance hierarchy, the major classes chosen when Ptolemy was first conceived — `Block`, `Star`, `Galaxy`, `Universe`, `PortHole`, and some others — still play the same roles. The first version had many public data members and did not use `const`; fortunately, these conditions were corrected before the project grew too large (though some public data members remain).

We find that Ptolemy is much easier to modify and extend than Lisp or C based systems of similar size; in fact, we are frequently reminded of this because one portion of Ptolemy, the portion of the graphical interface that connects to the VEM graphic editor, is written mostly in C, and it is far harder to modify than the rest of Ptolemy. The primary reason for this is type-testing: a different action must take place depending on the type of icon the mouse is pointing to when a command is executed, for example. In C++, this is handled neatly by virtual functions; in this particular C program, when a new type is added, modifications must be made at many points in the code. We have delayed rewriting of this code in C++ because of lack of resources, though we have probably paid the equivalent cost in extra maintenance time.

C++ provides no mechanism to accomplish certain repetitive tasks that are required in Ptolemy, particularly the following:

- We wish to refer to certain objects (states, in particular) through user interfaces by the same names as are used by the programmer in writing the code, so that if the user specifies an illegal value for state ‘‘bar’’, which is in star ‘‘foo’’ in galaxy ‘‘fred’’, the error message can refer to ‘‘fred.foo.bar’’. This is easily accomplished by associating a string ‘‘bar’’ with the state named bar, but this is highly error-prone; every name must be typed at least twice.
- We create new instances of stars and galaxies by cloning, so that each star must redefine the `clone` method to call its copy constructor, as follows:

```
Block* MyPersonalBlock::clone() const
{
        return new MyPersonalBlock(*this);
}
```

We wrote a preprocessor to allow the user to avoid typing in the standard ‘‘boilerplate’’ code (to provide names for objects and the clone method) when defining new stars; we also extended the preprocessor to generate documentation for the star in addition to its C++ code. This has been invaluable in producing an accurate, up-to-date manual for the system. Still, we were disappointed that the template proposal of [Ell90] is not powerful enough to generate the `clone` method as discussed above.

Because objects are created by a method that returns nothing but a pointer to `Block`, and because entirely new classes can be created by incremental linking, it is clearly not possible to do all type-checking at runtime. There are two places where a runtime type check may be necessary:

we must check that the stars in a galaxy are compatible with the current domain, and we must check, for stars that process packets, that a packet received by the star is of a type that can be processed by that star. For this purpose we use an `isA` method: the `isA` member function returns true if it is given the name of the class, or of any base class, of an object for which it is defined. Should `isA` return true, we then cast the pointer to the appropriate derived type. While there is some controversy in the C++ community about such things, in this case it is clearly necessary, though we have some concerns about the portability of the pointer downcasts in the scheme in the presence of multiple inheritance. We therefore welcome Stroustrup's proposal for `ptr_cast` and `ref_cast` in section 13.5 of [Str91], which, if accepted by the C++ community, provides a safer way of solving this problem.

The other uses of pointer casts in Ptolemy are primarily in container classes; we anticipate rewriting such classes to use templates when they become more commonly available.

We have done our best to use multiple inheritance as little as possible. We do not take a position as extreme as [Car91]; however, our early attempts to use multiple inheritance with virtual base classes convinced us that virtual base classes, at least, are to be avoided where possible, and, at least in our code, it has always been possible to do so. We find multiple inheritance much easier to deal with when virtual base classes are not involved, and we find single inheritance easier to deal with than multiple inheritance. We have restricted the use of multiple inheritance to situations where there are truly two ‘‘is-A’’ relationships to frequently used base classes.

7. CONCLUSIONS

The kernel of Ptolemy is fully implemented, and has been demonstrated in the context of numerous small examples. While many enhancements to this kernel are planned, the focus of the simulation portion of the Ptolemy project has now shifted to demonstrating a number of applications, and extensions to rapid prototyping. Currently, a large simulation of a multimedia packet speech and video network is under way, both to exercise Ptolemy and to show its utility. Further, the signal processing code generation capabilities of Gabriel are being re-implemented in Ptolemy, and additional code generation work that generalizes the SDF code generation model that was used in Gabriel and for embedded microcontroller applications is being pursued. A number of other applications of hardware-software co-design are being explored, including software for signaling and control of telecommunications switches. One of the goals is to demonstrate rapid prototyping of systems having hardware and software components, using the reusable software module approach that is natural within Ptolemy.

8. ACKNOWLEDGMENTS

Other than the authors, key contributors to Ptolemy include Ichiro Kuroda of NEC, who contributed to the development of the state handling mechanism, Edwin Goei, who contributed to the graphical interface, Wan-Teh Chang. who enhanced the graphical interface, and Andrea Casotto, who 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.

A variety of sponsors have directly or indirectly supported Ptolemy, including the Defense Advanced Projects Agency (DARPA), the Semiconductor Research Corporation (SRC), the National Science Foundation, the State of California MICRO Program, Motorola, Hughes Network Systems, Bell Northern Research, Sony, Hitachi, NEC, US West, and Comdisco Systems. Many other sponsors have contributed indirectly by supporting Gabriel, the predecessor.

9. REFERENCES

[Car91] T. A. Cargill, "Controversy: The Case Against Multiple Inheritance in C++", *Computing Systems*, vol 4, #1, Winter 1991.

[Dor90] S. M. Dorward, R. Sethi, J. E. Shapiro, "Adding New Code to a Running C++ Program", *Usenix C++ Conference Proceedings*, Apr. 1990.

[Ell90] M. A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, 1990.

[Fab] L. James Faber, "Capsim", ECE Dept., North Carolina State University, Raleigh, NC 27695.

[Har89] D. Harrison, "VEM - Graphics Environment for Oct Designs", in *Oct Tools Distribution 3.0*, (R. Spickelmier, ed.), Electronics Research Laboratory, UC Berkeley, 1989.

[Lee87a] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers*, January 1987, **C-36(2)**.

[Lee89a] E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP", IEEE Trans. on Acoustics, Speech, and Signal Processing, Nov., 1989.

[Mes84a] D.G. Messerschmitt, "A Tool for Structured Functional Simulation", *IEEE Transactions on Selected Areas in Communications*, Jan. 1984.

[Mes84b] D. G. Messerschmitt, "Structured Interconnection of Signal Processing Programs" *Proceedings of Globecom 84*, Atlanta, GA, Dec., 1984.

[Str86] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986.

[Str91] B. Stroustrup, *The C++ Programming*, Second Edition, Addison Wesley, 1991.

[Thor] "Thor User's Guide", XCAD Corporation, 1989.