DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

OVERVIEW OF THE PTOLEMY PROJECT

Edward A. Lee, P.I.
eal@eecs.berkeley.edu

UCB/ERL Memorandum M98/72

FEBRUARY 1, 1999

1. Modeling and Design

The Ptolemy project studies heterogeneous modeling and design of concurrent systems. The focus is on *embedded systems*, particularly those that mix technologies, including for example analog and digital electronics, hardware and software, and electronics and mechanical devices (including MEMS, microelectromechanical systems). The focus is also on systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces.

Modeling is the act of representing a system or subsystem formally. A model might be mathematical, in which case it can be viewed as a set of assertions about properties of the system such as its functionality or physical dimensions. A model can also be constructive, in which case it defines a computational procedure that mimics a set of properties of the system. Constructive models are often used to describe behavior of a system in response to stimulus from outside the system. Constructive models are also called executable models.

Design is the act of defining a system or subsystem. Usually this involves defining one or more models of the system and refining the models until the desired functionality is obtained within a set of constraints.

Design and modeling are obviously closely coupled. In some circumstances, models may be immutable, in the sense that they describe subsystems, constraints, or behaviors that are externally imposed on a design. For instance, they may describe a mechanical system that is not under design, but must be controlled by an electronic system that is under design.

Executable models are sometimes called *simulations*, an appropriate term when the executable model is clearly distinct from the system it models. However, in many electronic systems, a model that starts as a simulation mutates into a software implementation of the system. The distinction between the model and the system itself becomes blurred in this case. This is particularly true for embedded software.

Embedded software is software that resides in devices that are not first-and-foremost computers. It is pervasive, appearing in automobiles, telephones, pagers, consumer electronics, toys, aircraft, trains, security systems, weapons systems, printers, modems, copiers, thermostats, manufacturing systems, appliances, etc. A technically active person probably interacts regularly with more pieces of embedded software than conventional software.

> *A major emphasis in the Ptolemy project is on the methodology for defining and producing embedded software together with the systems within which it is embedded.*

Executable models are constructed under a *model of computation,* which is the set of “laws of physics” that govern the interaction of components in the model. If the model is describing a mechanical system, then the model of computation may literally be the laws of physics. More commonly, however, it is a set of rules that are more abstract, and provide a framework within which a designer builds models. A set of rules that govern the interaction of components is called the *semantics* of the model of computation. A model of computation may have more than one semantics, in that there might be distinct sets of rules that impose identical constraints on behavior.

The choice of model of computation depends strongly on the type of model being constructed. For example, for a purely computational system that transforms a finite body of data into another finite body of data, the imperative semantics that is common in programming languages such as C, C++, Java, and Matlab will be adequate. For modeling a mechanical system, the semantics needs to be able to handle concurrency and the time continuum, in which case a continuous-time model of computation such that found in Simulink, Saber, Hewlett-Packard’s ADS, and VHDL-AMS is more appropriate.

The ability of a model to mutate into an implementation depends heavily on the model of computation that is used. Some models of computation, for example, are suitable for implementation only in customized hardware, while others are poorly matched to customized hardware because of their intrinsically sequential nature. Choosing an inappropriate model of computation may compromise the quality of design by leading the designer into a more costly or less reliable implementation.

> *A principle of the Ptolemy project is that the choices of models of computation strongly affect the quality of a system design.*

For embedded systems, the most useful models of computation handle concurrency and time. This is because embedded systems consist typically of components that operate simultaneously and have multiple simultaneous sources of stimuli. In addition, they operate in a timed (real world) environment, where the timeliness of their response to stimuli may be as important as the correctness of the response.

> *The objective in Ptolemy II is to support the construction and interoperability of executable models that are built under a wide variety of models of computation.*

2. Models of Computation

There are a rich variety of models of computation that deal with concurrency and time in different ways. In this section, we outline some of the most useful models for embedded systems. All of these will lend a semantics to the same bubble-and-arc, or block-and-arrow diagram shown in figure 2.1.

2.1 Differential Equations

One possible semantics for the syntax in figure 2.1 is that of differential equations. The arcs represent continuous functions of a continuum that is interpreted as time. The bubbles represent relations between these functions. The job of a simulator is to find a fixed-point, i.e., a set of functions that satisfy all the relations.

Differential equations are excellent for modeling analog circuits and many physical systems. This is the model of computation used in Simulink, Saber, and VHDL-AMS, and is closely related to that in Spice circuit simulators. However, they have disadvantages. Since they directly describe a physical system, they are tightly bound to an implementation, leaving few implementation options. Moreover, they are only applicable to relatively well-understood technologies, where lumped-parameter modeling is appropriate. They must be generalized to partial differential equations for less understood technologies, where solution techniques such as finite elements can be quite costly. For well-understood technologies, they can be expensive to simulate compared to digital representations of comparable functionality (and hence, they can be expensive to implement in software).

Embedded systems frequently contain components that are best modeled using differential equations, such as MEMS and other mechanical components, analog circuits, and microwave circuits. These components, however, interact with an electronic system that may serve as a controller or a recipient of sensor data. This electronic system may be digital, in which case there is a fundamental mismatch in models of computation. Joint modeling of a continuous subsystem with digital electronics is known as *mixed signal modeling*.

2.2 Difference Equations

Differential equations can be discretized to get difference equations, a commonly used model of computation in digital signal processing. This model of computation can be further generalized to support multirate difference equations. In either case, a global *clock* defines the discrete points at which signals have values (at the *ticks*).

Difference equations are considerably easier to implement in software, and hence leave more freedom of implementation. Their key weaknesses are the global synchronization implied by the clock, and the awkwardness of specifying irregularly timed events and control logic.

The *synchronous dataflow* (SDF) domain in Ptolemy II is extended with a model of time to model difference equations. Dataflow models are discussed below in section 2.7.

2.3 Finite-State Machines

In FSMs, bubbles represent system *state* and arcs represent state *transitions*. The simple FSM model of computation is not concurrent. Execution is a strictly ordered sequence of state transitions.

FSM models are excellent for control logic in embedded systems, particularly safety-critical systems. FSM models are amenable to in-depth formal analysis, and thus can be used to avoid surprising

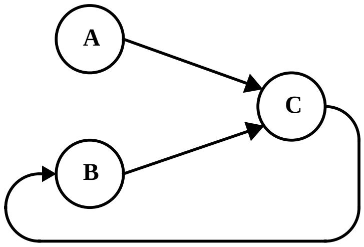

FIGURE 2.1. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

behavior. Moreover, FSMs are easily mapped to either hardware or software implementations.

FSM models have a number of key weaknesses. First, at a very fundamental level, they are not as expressive as the other models of computation described here. They are not sufficiently rich to describe all partially recursive functions. However, this weakness is acceptable in light of the formal analysis that becomes possible. Many questions about designs are decidable for FSMs and undecidable for other models of computation. A second key weakness is that the number of states can get very large even in the face of only modest complexity. This makes the models unwieldy.

The latter problem can often be solved by using FSMs in combination with concurrent models of computation. This was first noted by David Harel, who introduced that Statecharts formalism. Statecharts combine a loose version of synchronous-reactive modeling (described below) with FSMs [8]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [9].

A major (ongoing) result of the Ptolemy project has been to show that FSMs can be hierarchically combined with a huge variety of concurrent models of computation. We call the resulting formalism “*charts” (pronounced “starcharts”) where the star represents a wildcard [7].

2.4 Synchronous/Reactive Models

In the synchronous/reactive (SR) model of computation [1], the arcs represent data values that are aligned with global clock ticks. Thus, they are discrete signals, as with difference equations, but unlike difference equations, a signal need not have a value at every clock tick. The bubbles represent relations between input and output values at each tick, and are usually partial functions with certain technical restrictions to ensure determinacy. Examples of languages that use the SR model of computation include Esterel [3], Signal [2], Lustre [5], and Argos [16].

SR models are excellent for applications with concurrent and complex control logic. Because of the tight synchronization, safety-critical real-time applications are a good match. However, also because of the tight synchronization, some applications are overspecified in the SR model, limiting the implementation alternatives. Moreover, in most realizations, modularity is compromised by the need to seek a global fixed point at each clock tick.

2.5 Discrete-Event Models

In discrete-event (DE) models of computation, the arcs represent sets of *events* placed in time. An event consists of a *value* and *time stamp*. This model of computation is popular for specifying hardware and simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog. Unlike the SR model, there is no global clock tick, but like SR, differential equations, and difference equations, there is a globally consistent notion of time.

DE models are excellent descriptions of concurrent hardware, although increasingly the globally consistent notion of time is problematic. In particular, it over-specifies (or over-models) systems where maintaining such a globally consistent notion is difficult, including large VLSI chips with high clock rates. A key weakness is that it is relatively expensive to implement in software, as evidenced by the relatively slow simulators.

2.6 Synchronous Message Passing

In synchronous message passing, processes communicate in atomic, instantaneous actions called *rendezvous*. If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate. “Atomic” means that the two processes are simultaneously involved in the exchange, and that the exchange is initiated and completed in a single uninterruptable step. Examples of rendezvous models include Hoare’s *communicating sequential processes* (CSP) [11]and Milner’s *calculus of communicating systems* (CCS) [19]. This model of computation has been realized in a number of concurrent programming languages, including Lotos and Occam.

Rendezvous models are particularly well-matched to applications where resource sharing is a key element, such as client-server database models and multitasking or multiplexing of hardware resources. A key weakness of rendezvous-based models is that maintaining determinacy can be difficult. Proponents of the approach, of course, cite the ability to model nondeterminacy as a key strength.

2.7 Asynchronous Message Passing

In asynchronous message passing, processes communicate by sending messages through channels that can buffer the messages. The sender of the message need not wait for the receiver to be ready to receive the message. There are several variants of this technique, but we focus on those that ensure determinate computation, namely Kahn process networks [12] and dataflow models.

In a process network (PN) model of computation, the arcs represent sequences of data values (tokens), and the bubbles represent functions that map input sequences into output sequences. Certain technical restrictions on these functions are necessary to ensure determinacy, meaning that the sequences are fully specified. Dataflow models, popular in signal processing, are a special case of process networks [14].

PN models are excellent for signal processing. They are loosely coupled, and hence relatively easy to parallelize or distribute. They can be implemented efficiently in both software and hardware, and hence leave implementation options open. A key weakness of PN models is that they are awkward for specifying control logic.

Several special cases of PN are useful in certain circumstances. Dataflow models construct processes of a process network as sequences of atomic actor *firings*. Synchronous dataflow (SDF) is a particularly restricted special case with the extremely useful property that deadlock and boundedness are decidable. Boolean dataflow (BDF) is a generalization that sometimes yields to deadlock and boundedness analysis, although fundamentally these questions are undecidable. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness. The general case, process networks (PN), is implemented in Ptolemy II using Java threads for the processes.

2.8 Timed CSP and Timed PN

CSP and PN both involve threads that communicate via message passing, synchronously in the former case and asynchronously in the latter. Neither model intrinsically includes a notion of time, which can make it difficult to interoperate with models that do include a notion of time. In fact, message events are partially ordered, rather than totally ordered as they would be were they placed on a time line.

Both models of computation can be augmented with a notion of time to promote interoperability. Threads assume that time does not advance while they are active, but can advance when they stall on inputs, outputs, or explicitly indicate that time can advance. By this vehicle, additional constraints are imposed on the order of events, and determinate interoperability with timed models of computation becomes possible.

3. Choosing Models of Computation

The rich variety of concurrent models of computation outlined in the previous section can be daunting to a designer faced with having to select them. Most designers today do not face this choice because they get exposed to only one or two. This is changing, however, as the level of abstraction and domain-specificity of design software both rise. We expect that sophisticated and highly visual user interfaces will be needed to enable designers to cope with this heterogeneity.

An essential difference between concurrent models of computation is their modeling of time. Some are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line. Others are more abstract and take time to be discrete. Others are still more abstract and take time to be merely a constraint imposed by causality. This latter interpretation results in time that is partially ordered, and explains much of the expressiveness in process networks and rendezvous-based models of computation. Partially ordered time provides a mathematical framework for formally analyzing and comparing models of computation [15].

A grand unified approach to modeling would seek a concurrent model of computation that serves all purposes. This could be accomplished by creating a *melange*, a mixture of all of the above, but such a mixture would be extremely complex and difficult to use, and synthesis and simulation tools would be difficult to design.

Another alternative would be to choose one concurrent model of computation, say the rendezvous model, and show that all the others are subsumed as special cases. This is relatively easy to do, in theory. Most of these models of computation are sufficiently expressive to be able to subsume most of the others. However, this fails to acknowledge the strengths and weaknesses of each model of computation. Differential equations, for instance, are very good at describing the interaction of point masses in a model of a MEMS system, but not as good at describing the discrete control logic that may be ultimately controlling the actuators in the MEMS system. Similarly, finite-state machines are good at modeling at least simple control logic, but hopelessly inadequate for modeling the interaction of point masses. Thus, to design interesting systems, designers need to use heterogeneous models.

4. Visual Syntaxes

Visual depictions of electronic systems have always held a strong human appeal, making them extremely effective in conveying information about a design. Many of the domains of interest in the Ptolemy project use such depictions to completely and formally specify models.

> *One of the principles of the Ptolemy project is that visual depictions of systems can help to offset the increased complexity that is introduced by heterogeneous modeling.*

These visual depictions offer an alternative *syntax* to associate with the semantics of a model of computation. Visual syntaxes can be every bit as precise and complete as textual syntaxes, particularly when they are judiciously combined with textual syntaxes.

Visual representations of models have a mixed history. In circuit design, schematic diagrams used to be routinely used to capture all of the essential information needed to implement some systems. Schematics are often replaced today by text in hardware description languages such as VHDL or Verilog. In other contexts, visual representations have largely failed, for example flowcharts for capturing the behavior of software. Recently, a number of innovative visual formalisms have been garnering support, including visual dataflow, hierarchical concurrent finite state machines, and object models. The UML visual language for object modeling has been receiving a great deal of attention, and in fact is used fairly extensively in the design of Ptolemy II itself.

A subset of visual languages that are recognizable as “block diagrams” represent concurrent systems. There are many possible concurrency semantics (and many possible models of computation) associated with such diagrams. Formalizing the semantics is essential if these diagrams are to be used for system specification and design. Ptolemy II supports exploration of the possible concurrency semantics. A principle of the project is that the strengths and weaknesses of these alternatives make them complementary rather than competitive. Thus, interoperability of diverse models is essential.

5. Ptolemy II

Ptolemy II is a complete, from the ground up, redesign of the Ptolemy 0.x software environment [4], which supports heterogeneous modeling and design of concurrent systems. It offers a unified infrastructure for implementations of a number of models of computation. The overall architecture consists of a set of packages that provide generic support for all models of computation and a set of packages that provide more specialized support for particular models of computation. Examples of the former include packages that contain math libraries, graph algorithms, an interpreted expression language, signal plotters, and interfaces to media capabilities such as audio. Examples of the latter include packages that support clustered graph representations of models, packages that support executable models, and *domains*, which are packages that implement a particular model of computation.

5.1 Package Structure

The package structure is shown in figure 5.1. This is a UML package diagram [22]. The name of each package is in the tab at the top of each box. Subpackages are contained within their parent package. Dependencies between packages are shown by dotted lines with arrow heads. For example, *actor* depends on *kernel.event* which depends on *kernel* which depends on *kernel.util. Actor* also depends on *data* and *graph*. The role of each package is explained below.

actor This package supports executable entities that receive and send data through ports. It includes both untyped and typed actors. For typed actors, it implements a sophisticated type system that supports polymorphism. It includes the base class Director for domain-specific classes that control the execution of a model.

actor.lib This subpackage is a library of polymorphic actors.

actor.process This subpackage provides infrastructure for domains where actors are processes implemented on top of Java threads.

actor.sched This subpackage provides infrastructure for domains where actors are statically scheduled by the director.

actor.util This subpackage contains utilities that support directors in various domains. Specifically, it contains a simple FIFO Queue and a sophisticated priority queue called a calendar queue.

data This package provides classes that encapsulate and manipulate data that is transported between actors in Ptolemy models.

data.expr This class supports an extensible expression language and an interpreter for that language. Parameters can have values specified by expressions. These expressions

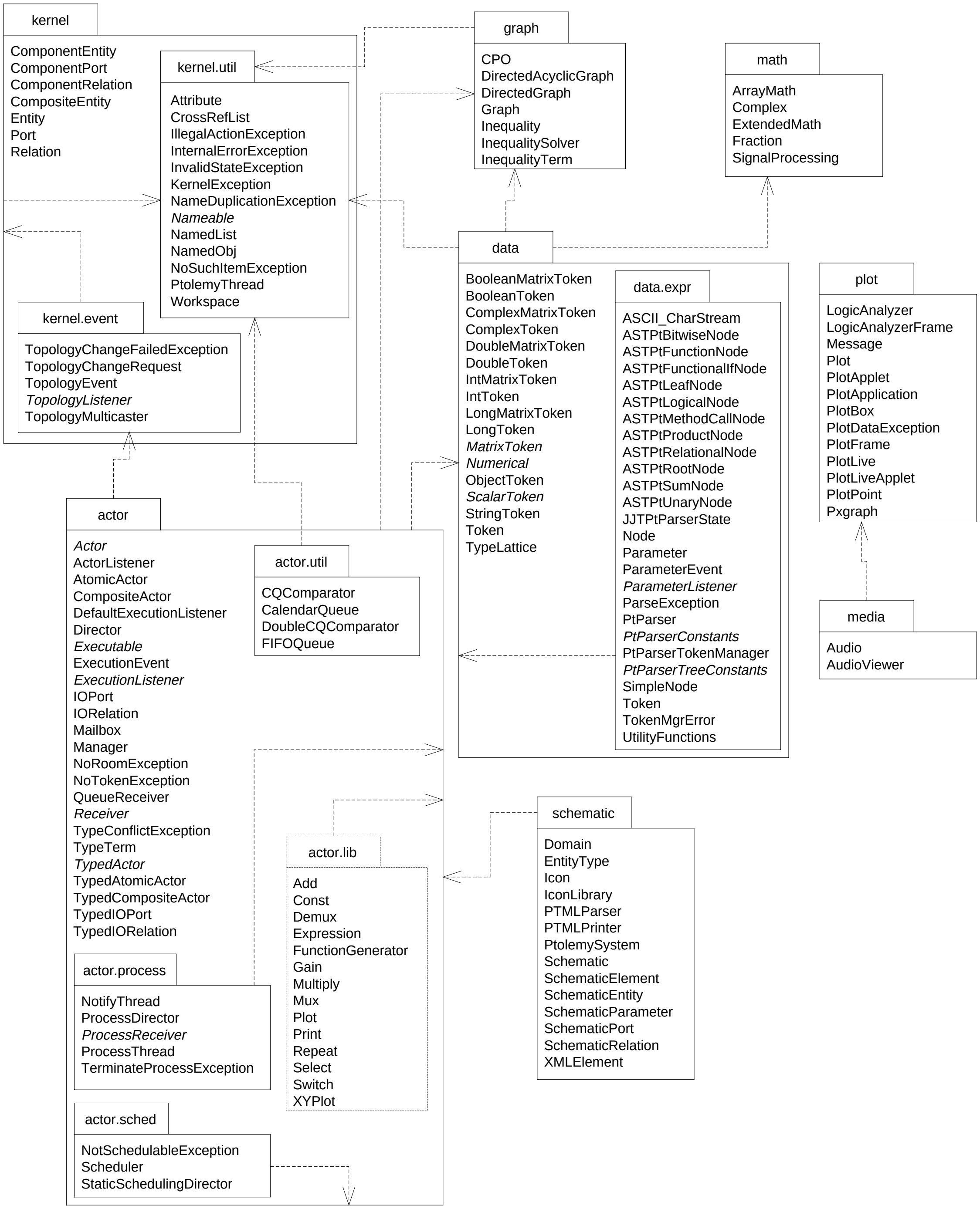

FIGURE 5.1. The package structure of Ptolemy II. The actor.lib package has not yet been fully constructed.

may refer to other parameters. Dependencies between parameters are handled transparently, as in a spreadsheet, where updating the value of one will result in the update of all those that depend on it.

graph This package provides algorithms for manipulating and analyzing mathematical graphs. Mathematical graphs are simpler than Ptolemy II clustered graphs in that there is no hierarchy, and arcs link exactly two nodes. This package is expected to supply a growing library of algorithms.

kernel This package provides the software architecture for the key abstract syntax, clustered graphs. The classes in this package support entities with ports, and relations that connect the ports. Clustering is where a collection of entities is encapsulated in a single composite entity, and a subset of the ports of the inside entities are exposed as ports of the cluster entity.

kernel.event This package contains classes and interfaces that support controlled mutations of clustered graphs. Mutations are modifications in the topology, and in general, they are permitted to occur during the execution of a model. But in certain domains, where maintaining determinacy is imperative, the director may wish to exercise tight control over precisely when mutations are performed. This package supports queueing of mutation requests for later execution. It uses a publish-and-subscribe design pattern.

kernel.util This subpackage of the kernel package provides a collection of utility classes that do not depend on the kernel package. It is separated into a subpackage so that these utility classes can be used without the kernel. The utilities include a collection of exceptions, classes supporting named objects with attributes, lists of named objects, a specialized cross-reference list class, and a thread class that helps Ptolemy keep track of executing threads.

math This package encapsulates mathematical functions and methods for operating on matrices and vectors. It also includes a complex number class and a class supporting fractions.

media This package encapsulates a set of classes supporting audio and image processing.

plot This package provides two-dimensional signal plotting widgets.

schematic This package provides a top-level interface to Ptolemy II. A GUI can use the classes in this package to gain access to Ptolemy II models.

5.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 5.2. This is a *static structure diagram* in UML (unified modeling language). The key syntactic elements are boxes, which represent classes, the hollow arrow, which indicates generalization, and other lines, which indicate association. Some lines have a small diamond, which indicates aggregation.

Instances of all of the classes shown can have names; they all implement the Nameable interface. Most of the classes generalize NamedObj, which in addition to being nameable can have a list of attributes associated with it. Attributes themselves are instances of NamedObj.

Entity, Port, and Relation are three key classes that extend NamedObj. These classes define the primitives of the abstract syntax supported by Ptolemy II. They will be fully explained in the kernel chapter. ComponentPort, ComponentRelation, and ComponentEntity extend these classes by adding support for clustered graphs. CompositeEntity extends ComponentEntity and represents an aggregation of instances of ComponentEntity and ComponentRelation.

The Executable interface defines objects that can be executed. The Actor interface extends this with capability for transporting data through ports. AtomicActor and CompositeActor are concrete classes that implement this interface.

An executable Ptolemy II model consists of a top-level CompositeActor with an instance of Director and an instance of Manager associated with it. The manager provides overall control of the execution (starting, stopping, pausing). The director implements a semantics of a model of computation to govern the execution of actors contained by the CompositeActor.

Director is the base class for directors that implement models of computation. Each such director is associated with a domain. We have defined in Ptolemy II directors that implement continuous-time modeling (ODE solvers), process networks, synchronous dataflow, discrete-event modeling, and communicating sequential processes.

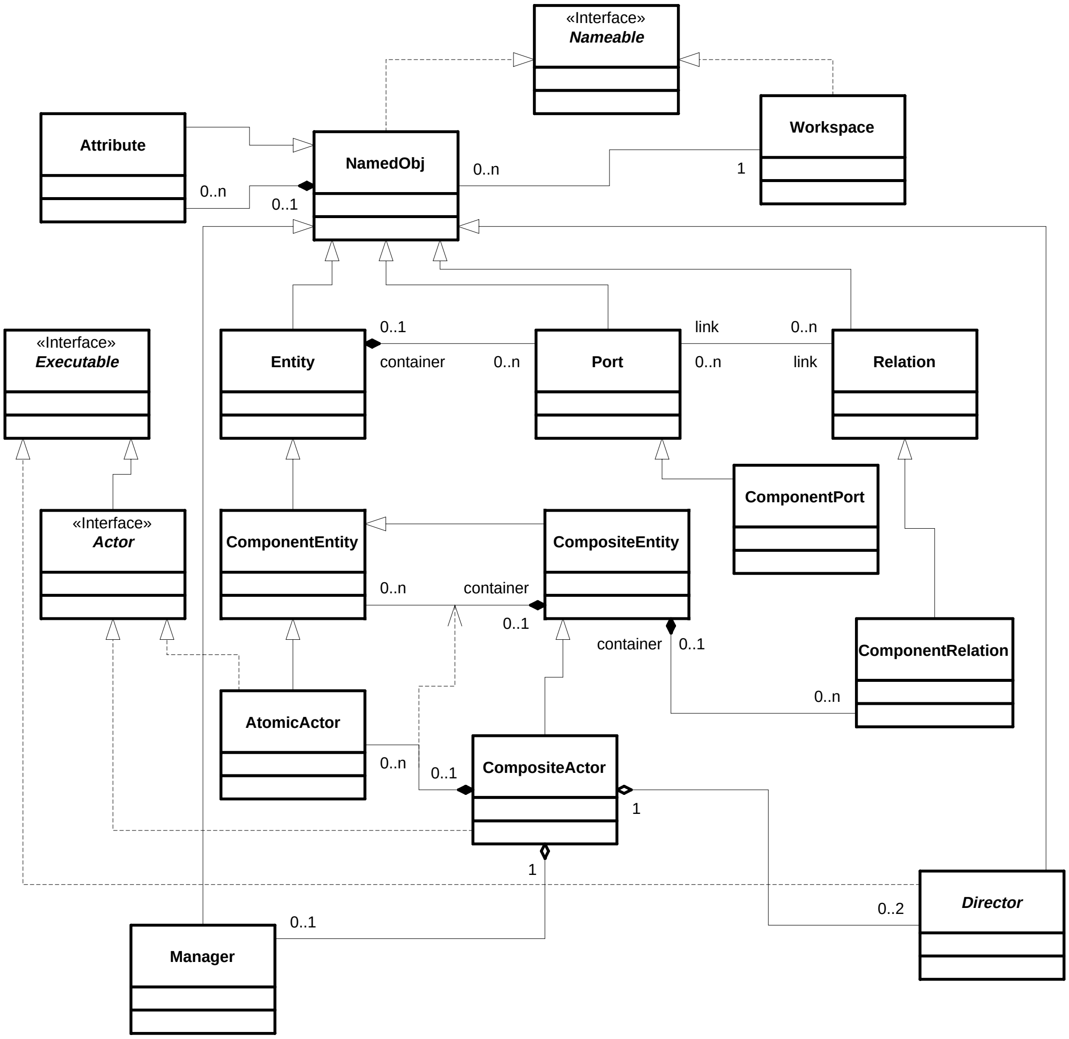

FIGURE 5.2. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and *actor* packages.

5.3 Capabilities

Ptolemy II is a second generation system. Its predecessor, Ptolemy 0.x, still has many active users and developers, and may continue to evolve for some time. Ptolemy II has a somewhat different emphasis, and through its use of Java, concurrency, and integration with the network, is aggressively experimental. Some of the major capabilities in Ptolemy II that we believe to be new technology in modeling and design environments include:

- *Higher level concurrent design in Java^{TM}*. Java support for concurrent design is very low level, based on threads and monitors. Maintaining safety and liveness can be quite difficult [13]. Ptolemy II includes a number of domains that support design of concurrent systems at a much higher level of abstraction. These include, at varying levels of maturity, process networks, communicating sequential processes (rendezvous based), dataflow, synchronous/reactive modeling, continuous-time modeling, and hierarchical concurrent finite-state machines.
- *Better modularization through the use of packages*. Ptolemy II is divided into packages that can be used independently and distributed on the net, or drawn on demand from a server. This breaks with tradition in design software, where tools are usually embedded in huge integrated systems with interdependent parts.
- *Complete separation of the abstract syntax from the semantics*. Ptolemy designs are structured as clustered graphs. Ptolemy II defines a clean and thorough abstract syntax for such clustered graphs, and separates into distinct packages the infrastructure supporting such graphs from mechanisms that attach semantics (such as dataflow, analog circuits, finite-state machines, etc.) to the graphs.
- *Improved heterogeneity*. Previous realizations of Ptolemy provided a wormhole mechanism for hierarchically coupling heterogeneous models of computation. This mechanism is improved in Ptolemy II through the use of opaque composite actors, which provide better support for models of computation that are very different from dataflow, the best supported model in prior versions of Ptolemy software. These include hierarchical concurrent finite-state machines and continuous-time modeling techniques.
- *Thread-safe concurrent execution*. Ptolemy models are typically concurrent, but in the past, support for concurrent execution of a Ptolemy model has been primitive. Ptolemy II supports concurrency throughout, allowing for instance for a model to mutate (modify its clustered graph structure) while the user interface simultaneously modifies the structure in different ways. Consistency is maintained through the use of monitors and read/write semaphores [11] built upon the lower level synchronization primitives of Java.
- *A software architecture based on object modeling*. Since the first Ptolemy implementation, software engineering has seen the emergence of sophisticated object modeling [18][23][25] and design pattern [6] concepts. We have applied these concepts to the design of Ptolemy II, and they have resulted in a more consistent, cleaner, and more robust design. We have also applied a simplified software engineering process that includes systematic design and code reviews [17][21].
- *A truly polymorphic type system*. Earlier implementations of Ptolemy supported rudimentary polymorphism through the “anytype” particle. Even with such limited polymorphism, type resolution proved challenging, and the implementation is ad-hoc and fragile. Ptolemy II has a more modern type system based on a partial order of types and monotonic type refinement functions associated with functional blocks. Type resolution consists of finding a fixed point, using algorithms inspired by the type system in ML [20].
- *Domain-polymorphic actors*. In earlier implementations of Ptolemy, actor libraries were separated

by domain. Through the notion of subdomains, actors could operate in more than one domain. In Ptolemy II, this idea is taken much further. Actors with intrinsically polymorphic functionality can be written to operate in a much larger set of domains. The mechanism they use to communicate with other actors depends on the domain in which they are used. This is managed through a concept that we call a **process level type system**.

5.4 Future Capabilities

Capabilities that we anticipate making available in the future include:

- *Extensible XML-based file formats*. XML is an emerging standard for representation of information that focuses on the logical relationships between pieces of information. Human-readable representations are generated with the help of style sheets. Ptolemy II will use XML as its primary format for persistent design data.
- *Interoperability through software components*. Ptolemy II will use distributed software component technology such as CORBA, JINI, or COM, in a number of ways. Components (actors) in a Ptolemy II model will be implementable on a remote server. Also, components may be parameterized where parameter values are supplied by a server (this mechanism supports *reduced-order modeling*, where the model is provided by the server). Ptolemy II models will be exported via a server. And finally, Ptolemy II will support migrating software components.
- *Embedded software synthesis.* Pertinent Ptolemy II domains will be tuned to run on a Java virtual machine on an embedded CPU. Hardware, firmware, and configurable hardware components will expose abstractions to this Java software that obey the model of computation of the pertinent domain. Java's native code interface will be used to define a stub for the embedded hardware components so that they are indistinguishable from any other Java thread within the model of computation. Domains that seem particularly well suited to this approach include PN and CSP.
- *Embedded hardware synthesis*. Earlier versions of Ptolemy had only very weak mechanisms for migrating designs from idealized floating-point simulations through fixed-point simulations to embedded software, FPGA, and hardware designs. Ptolemy II will separate the interface definition of component blocks from their implementation, allowing libraries to be constructed where compatibility across implementation technologies is assured [24]. This work is currently being prototyped in Ptolemy 0.7.1.
- *Integrated verification tools.* Modern verification tools based on model checking [10] could be integrated with Ptolemy II at least to the extent that finite state machine models can be checked. We believe that the separation of control logic from concurrency will greatly facilitate verification, since only much smaller cross-sections of the system behavior will be offered to the verification tools.

6. References

[1] A.. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[2] A. Benveniste and P. Le Guernic, “Hybrid Dynamical Systems Theory and the SIGNAL Language,” *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

[3] G. Berry and G. Gonthier, “The Esterel synchronous programming language: Design, semantics, implementation,” *Science of Computer Programming*, 19(2):87-152, 1992.

[4] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation*, special issue on “Simulation Software Development,” vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/papers/JEurSim).

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, “LUSTRE: A Declarative Language for Programming Synchronous Systems,” *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA, 1995.

[7] A. Girault, B. Lee, and E. A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models,” April 13, 1998 (revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, CA 94720, August 1997).
(http://ptolemy.eecs.berkeley.edu/papers/98/starcharts)

[8] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[9] T. A. Henzinger, “The theory of hybrid automata,” in *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1996, pp. 278-292, invited tutorial.

[10] T.A. Henzinger, and O. Kupferman, and S. Qadeer, “From *pre*historic to *post*modern symbolic model checking,” in *CAV 98: Computer-aided Verification*, pp. 195-206, eds. A.J. Hu and M.Y. Vardi, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998.

[11] C. A. R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[12] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[13] D. Lea, *Concurrent Programming in Java^{TM}*, Addison-Wesley, Reading, MA, 1997.

[14] E. A. Lee and T. M. Parks, “Dataflow Process Networks,”, *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)

[15] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,”, March 12, 1998. (Revised from ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997).
(http://ptolemy.eecs.berkeley.edu/papers/98/framework/)

[16] F. Maraninchi, “The Argos Language: Graphical Representation of Automata and Description of Reactive Systems,” in *Proc. of the IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[17] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993.

[18] B. Meyer, *Object Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

[19] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[20] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[21] NASA Office of Safety and Mission Assurance, *Software Formal Inspections Guidebook*, August 1993 (http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt).

[22] Rational Software Corporation, *UML Notation Guide*, Version 1.1, September 1997, http://www.rational.com/uml/html/notation/.

[23] A. J. Riel, *Object Oriented Design Heuristics*, Addison Wesley, 1996.

[24] J. Rowson and A. Sangiovanni-Vincentelli, “Interface Based Design,” *Proc. of DAC ‘97*.

[25] J. Rumbaugh, *et al*. *Object-Oriented Modeling and Design* Prentice Hall, 1991.