OVERVIEW OF THE PTOLEMY PROJECT

by

Edward A. Lee, Principal Investigator
Christopher Hylands, Jorn Janneck, John Davis, II, Jie Liu,
Xiaojun Liu, Steve Neuendorffer, Sonia Sachs,
Mary Stewart, Kees Vissers, Paul Whitaker and
Yuhong Xiong

OVERVIEW OF THE PTOLEMY PROJECT

by

Edward A. Lee, Principal Investigator
Christopher Hylands, Jorn Janneck, John Davis, II, Jie Liu,
Xiaojun Liu, Steve Neuendorffer, Sonia Sachs, Mary Stewart,
Kees Vissers, Paul Whitaker and Yuhong Xiong

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OVERVIEW OF THE PTOLEMY PROJECT

MARCH 6, 2001

http://ptolemy.eecs.berkeley.edu/

John Davis, II
Christopher Hylands
Jörn Janneck
Edward A. Lee, Principal Investigator
Jie Liu
Xiaojun Liu
Steve Neuendorffer
Sonia Sachs
Mary Stewart
Kees Vissers
Paul Whitaker
Yuhong Xiong

1. Modeling and Design

The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. The focus is on *embedded systems* [21], particularly those that mix technologies, including for example analog and digital electronics, hardware and software, and electronics and mechanical devices. 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 Ptolemy II 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.

Ptolemy II takes a component view of design, in that models are constructed as a set of interacting components. A model of computation governs the semantics of the interaction, and thus imposes a discipline on the interaction of the interaction of components.

Component-based design in Ptolemy II involves disciplined interactions between components governed by a model of computation.

2. Architecture Design

Architecture description languages (ADLs), such as Wright [1] and Rapide [27], focus on formalisms for describing the rich sorts of component interactions that commonly arise in software architecture. Ptolemy II, by contrast, might be called an *architecture design language*, because its objective is not so much to describe existing interactions, but rather to promote coherent software architecture by imposing some structure on those interactions. Thus, while an ADL might focus on the compatibility of a sender and receiver in two distinct components, we would focus on a pattern of interactions among a set of components. Instead of, for example, verifying that a particular protocol in a single port-to-port interaction does not deadlock [1], we would focus on whether an assemblage of components can deadlock.

It is arguable that our approach is less modular, because components must be designed to the framework. Typical ADLs can describe pre-existing components, whereas in Ptolemy II, such pre-existing components would have to wrapped in Ptolemy II actors. Moreover, designing components to a particular interface may limit their reusability, and in fact the interface may not match their needs well. All of these are valid points, and indeed a major part of our research effort is to ameliorate these limitations. The net effect, we believe, is an approach that is much more powerful than ADLs.

First, we design components to be *domain polymorphic*, meaning that they can interact with other components within a wide variety of domains. In other words, instead of coming up with an ADL that can describe a number of different interaction mechanisms, we have come up with an architecture where components can be easily designed to interact in a number of ways. We argue that this makes the components more reusable, not less, because disciplined interaction within a well-defined semantics is possible. By contrast, with pre-existing components that have rigid interfaces, the best we can hope for is ad-hoc synthesis of adapters between incompatible interfaces, something that is likely to lead to designs that are very difficult to understand and to verify. Whereas ADLs draw an analogy between compatibility of interfaces and type checking [1], we use a technique much more powerful than type checking alone, namely polymorphism [24].

Second, to avoid the problem that a particular interaction mechanism may not fit the needs of a component well, we provide a rich set of interaction mechanisms embodied in the Ptolemy II domains. The domains force component designers to think about the overall pattern of interactions, and trade off uniformity for expressiveness. Where expressiveness is paramount, the ability of Ptolemy II to hierarchically mix domains offers essentially the same richness of more ad-hoc designs, but with much more discipline. By contrast, a non-trivial component designed without such structure is likely to use a *melange*, or ad-hoc mixture of interaction mechanisms, making it difficult to embedded it within a comprehensible system.

Third, whereas an ADL might choose a particular model of computation to provide it with a formal structure, such as CSP for Wright [1], we have developed a more abstract formal framework that

describes models of computation at a meta level [23]. This means that we do not have to perform awkward translations to describe one model of computation in terms of another. For example, stream based communication via FIFO channels are awkward in Wright [1].

We make these ideas concrete by describing the models of computation implemented in the Ptolemy II domains.

3. Models of Computation

There is a rich variety of models of computation that deal with concurrency and time in different ways. Each gives an interaction mechanism for components. In this section, we describe models of computation that are implemented in Ptolemy II domains, plus a couple of additional ones that are planned. Our focus has been on models of computation that are most useful for embedded systems. All of these can lend a semantics to the same bubble-and-arc, or block-and-arrow diagram shown in figure 1. Ptolemy II models are (clustered, or hierarchical) graphs of the form of figure 1, where the nodes are *entities* and the arcs are *relations*. For most domains, the entities are *actors* (entities with functionality) and the relations connecting them represent communication between actors.

3.1 Communicating Sequential Processes - CSP

In the CSP domain (communicating sequential processes), created by Neil Smyth [40], actors represent concurrently executing processes, implemented as Java threads. These processes communicate by atomic, instantaneous actions called *rendezvous* (or sometimes, *synchronous message passing*). 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) [17] and Milner's *calculus of communicating systems* (CCS) [31]. 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 feature of rendezvous-based models is their ability to cleanly model nondeterminate interactions. The CSP domain implements both conditional send and conditional receive. It also includes an experimental timed extension.
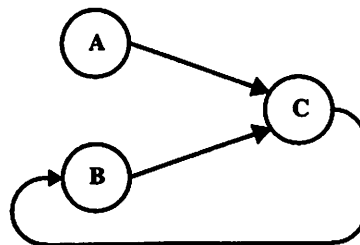

FIGURE 1. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

3.2 Continuous Time - CT

In the CT domain (continuous time), created Jie Liu [25], actors represent components that interact via continuous-time signals. Actors typically specify algebraic or differential relations between inputs and outputs. The job of the director in the domain is to find a fixed-point, i.e., a set of continuous-time functions that satisfy all the relations.

The CT domain includes an extensible set of differential equation solvers. The domain, therefore, is useful for modeling physical systems with linear or nonlinear algebraic/differential equation descriptions, such as analog circuits and many mechanical systems. Its model of computation is similar to that used in Simulink, Saber, and VHDL-AMS, and is closely related to that in Spice circuit simulators.

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. Joint modeling of a continuous subsystem with digital electronics is known as *mixed signal modeling* [26]. The CT domain is designed to interoperate with other Ptolemy domains, such as DE, to achieve mixed signal modeling. To support such modeling, the CT domain models of discrete events as Dirac delta functions. It also includes the ability to precisely detect threshold crossings to produce discrete events.

Physical systems often have simple models that are only valid over a certain regime of operation. Outside that regime, another model may be appropriate. A *modal model* is one that switches between these simple models when the system transitions between regimes. The CT domain interoperates with the FSM domain to create modal models.

3.3 Discrete-Events - DE

In the discrete-event (DE) domain, created by Lukito Muliadi, the actors communicate via sequences of events placed in time, along a real time line. An *event* consists of a *value* and *time stamp*. Actors can either be processes that react to events (implemented as Java threads) or functions that fire when new events are supplied. This model of computation is popular for specifying digital hardware and for simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog.

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. Every event is placed precisely on a globally consistent time line.

The DE domain implements a fairly sophisticated discrete-event simulator. DE simulators in general need to maintain a global queue of pending events sorted by time stamp (this is called a *priority queue*). This can be fairly expensive, since inserting new events into the list requires searching for the right position at which to insert it. The DE domain uses a calendar queue data structure [5] for the global event queue. A calendar queue may be thought of as a hashtable that uses quantized time as a hashing function. As such, both enqueue and dequeue operations can be done in time that is independent of the number of events in the queue.

In addition, the DE domain gives deterministic semantics to simultaneous events, unlike most competing discrete-event simulators. This means that for any two events with the same time stamp, the

order in which they are processed can be inferred from the structure of the model. This is done by analyzing the graph structure of the model for data precedences so that in the event of simultaneous time stamps, events can be sorted according to a secondary criterion given by their precedence relationships. VHDL, for example, uses delta time to accomplish the same objective.

3.4 Distributed Discrete Events - DDE

The distributed discrete-event (DDE) domain, created by John Davis [8], can be viewed either as a variant of DE or as a variant of PN (described below). Still highly experimental, it addresses a key problem with discrete-event modeling, namely that the global event queue imposes a central point of control on a model, greatly limiting the ability to distribute a model over a network. Distributing models might be necessary either to preserve intellectual property, to conserve network bandwidth, or to exploit parallel computing resources.

The DDE domain maintains a local notion of time on each connection between actors, instead of a single globally consistent notion of time. Each actor is a process, implemented as a Java thread, that can advance its local time to the minimum of the local times on each of its input connections. The domain systematizes the transmission of null events, which in effect provide guarantees that no event will be supplied with a time stamp less than some specified value.

3.5 Discrete Time - DT

The discrete-time (DT) domain, written by Chamberlain Fong [10], extends the SDF domain (described below) with a notion of time between tokens. Communication between actors takes the form of a sequence of tokens where the time between tokens is uniform. Multirate models, where distinct connections have distinct time intervals between tokens, are also supported. There is considerable subtlety in this domain when multirate components are used. The semantics is defined so that component behavior is always causal, in that outputs whose values depend on inputs are never produced at times prior to those of the inputs.

3.6 Finite-State Machines - FSM

The finite-state machine (FSM) domain, written by Xiaojun Liu, is radically different from the other Ptolemy II domains. The entities in this domain represent not actors but rather *state*, and the connections represent *transitions* between states. Execution is a strictly ordered sequence of state transitions. The FSM domain leverages the built-in expression language in Ptolemy II to evaluate *guards*, which determine when state transitions can be taken.

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 behavior.

FSM models have some 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 partial 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.

Both problems 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 [14]. FSMs

have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [15].

The FSM domain in Ptolemy II can be hierarchically combined with other domains. We call the resulting formalism "*charts" (pronounced "starcharts") where the star represents a wildcard [12]. Since most other domains represent concurrent computations, *charts model concurrent finite state machines with a variety of concurrency semantics. When combined with CT, they yield hybrid systems and modal models. When combined with SR (described below), they yield something close to State-charts. When combined with process networks, they resemble SDL [39].

3.7 Process Networks - PN

In the process networks (PN) domain, created by Mudit Goel [13], 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. This style of communication is often called asynchronous message passing. There are several variants of this technique, but the PN domain specifically implements one that ensures determinate computation, namely Kahn process networks [18].

In the PN model of computation, the arcs represent sequences of data values (tokens), and the entities 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. In particular, the function implemented by an entity must be *prefix monotonic*. The PN domain realizes a subclass of such functions, first described by Kahn and MacQueen [19], where *blocking reads* ensure monotonicity.

PN models 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, although much of this awkwardness may be ameliorated by combining them with FSM.

The PN domain in Ptolemy II has a highly experimental timed extension. This adds to the blocking reads a method for stalling processes until time advances. We anticipate that this timed extension will make interoperation with timed domains much more practical.

3.8 Synchronous Dataflow - SDF

The synchronous dataflow (SDF) domain, created by Steve Neuendorffer, handles regular computations that operate on streams. Dataflow models, popular in signal processing, are a special case of process networks (for the complete explanation of this, see [22]). 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. Moreover, the schedule of firings, parallel or sequential, is computable statically, making SDF an extremely useful specification formalism for embedded real-time software and for hardware.

Certain generalizations sometimes yield to similar analysis. Boolean dataflow (BDF) models sometimes yield 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. Neither a BDF nor DDF domain has yet been written in Ptolemy II. Process networks (PN) serves in the interim to handle computations that do not match the restrictions of SDF.

3.9 Synchronous/Reactive - SR

In the synchronous/reactive (SR) model of computation [2], the arcs represent data values that are aligned with global clock ticks. Thus, they are discrete signals, but unlike discrete time, a signal need not have a value at every clock tick. The entities 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 [4], Signal [3], Lustre [7], and Argos [28].

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. An SR domain has not yet been implemented in Ptolemy II, although the methods used by Stephen Edwards in Ptolemy Classic can be adapted to this purpose [9].

4. 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 [23].

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. It is the premise of Wright, for example [1]. 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. Rendezvous is very good at resource management, but very awkward for loosely coupled data-oriented computations. Asynchronous message passing is the reverse, where resource management is awkward, but data-oriented computations are natural[1]. Thus, to

---

1. Consider the difference between the telephone (rendezvous) and email (asynchronous message passing). If you are trying to schedule a meeting between four busy people, getting them all on a conference call would lead to a quick resolution of the meeting schedule. Scheduling the meeting by email could take several days, and may in fact never converge. Other sorts of communication, however, are far more efficient by email.

to design interesting systems, designers need to use heterogeneous models.

5. Visual Syntaxes

Visual depictions of 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.

Figures 2 and 3 show two different visual renditions of Ptolemy II models. Both renditions are constructed in Vergil, the visual editor framework in Ptolemy II. In figure 2, a Ptolemy II model is shown as a block diagram, which is an appropriate rendition for many discrete event models. In this particular example, records are constructed at the left by composing strings with integers representing a sequence number. The records are launched into a network that introduces random delay. The records may arrive at the right out of order, but the Sequence actor is used to re-order them using the sequence number.
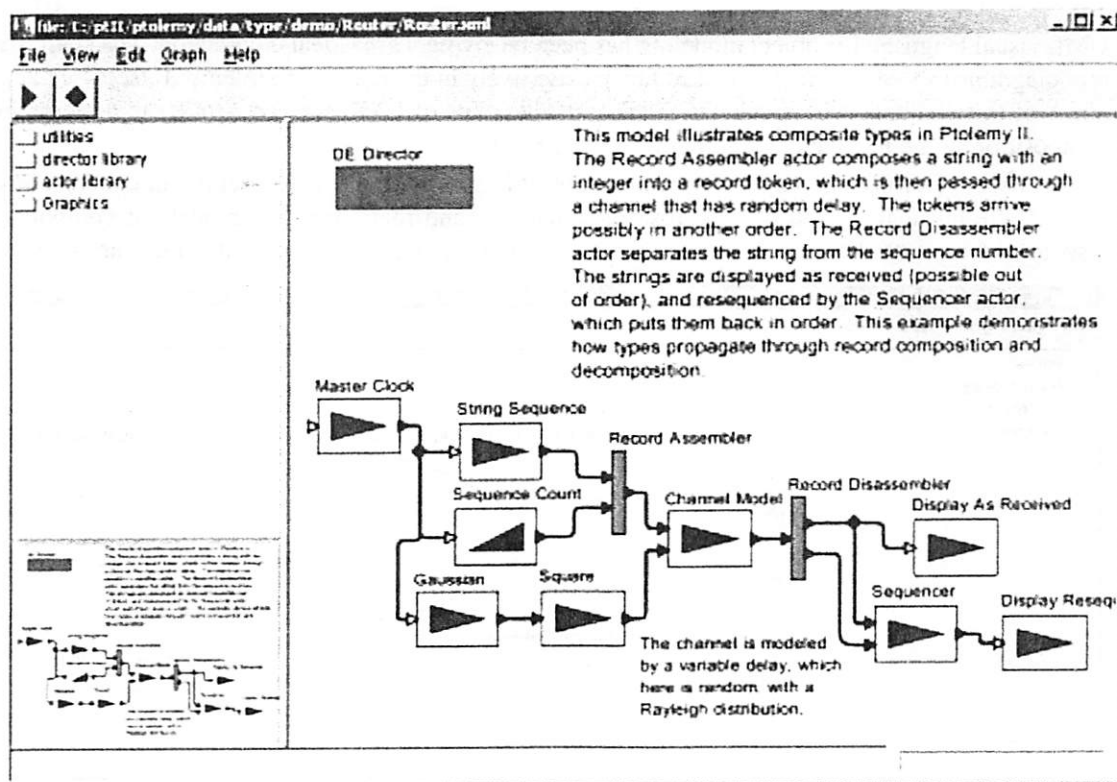
FIGURE 2. Visual rendition of a Ptolemy II model as a block diagram in Vergil (in the DE domain).

Figure 3 also shows a visual rendition of a Ptolemy II model, but now, the components are represented by circles, and the connections between components are represented by labeled arcs. This visual syntax is a familiar way to represent finite state machines (FSMs). Each circle represents a state of the model, and the arcs represent transitions between states. The particular example in the figure comes from a hybrid system model, where the two states, Separate and Together, represent two different modes of operation of a continuous-time system. The arcs are labeled with two lines, the first of which is a *guard*, and the second of which is an *action*. The guard is a boolean-valued expression that specifies when the transition should be taken, and the action is a sequence of commands that are executed when the transition is taken.

The visual renditions in figures 2 and 3 are both constructed using the same underlying infrastructure, Vergil, built by Stephen Neuendorffer. Vergil, in turn, in built on top of a GUI package called Diva, developed by John Reekie and Michael Shilman at Berkeley. Diva, in turn, is built on top of Swing and Java 2D, which are part of the Java platform from Sun Microsystems. In Vergil, a visual editor is constructed as an assembly of components in a Ptolemy II model. Thus, the system is configurable and customizable, and a great deal of infrastructure can be shared between the two distinct visual editors of figures 2 and 3.

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. The static structure diagrams of UML, in fact, are used fairly extensively in the design of Ptolemy II itself (see appendix A of this chapter). Moreover, the Statecharts diagrams of UML are very similar to a hierarchical composition of the FSM and SR domains in Ptolemy II.

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
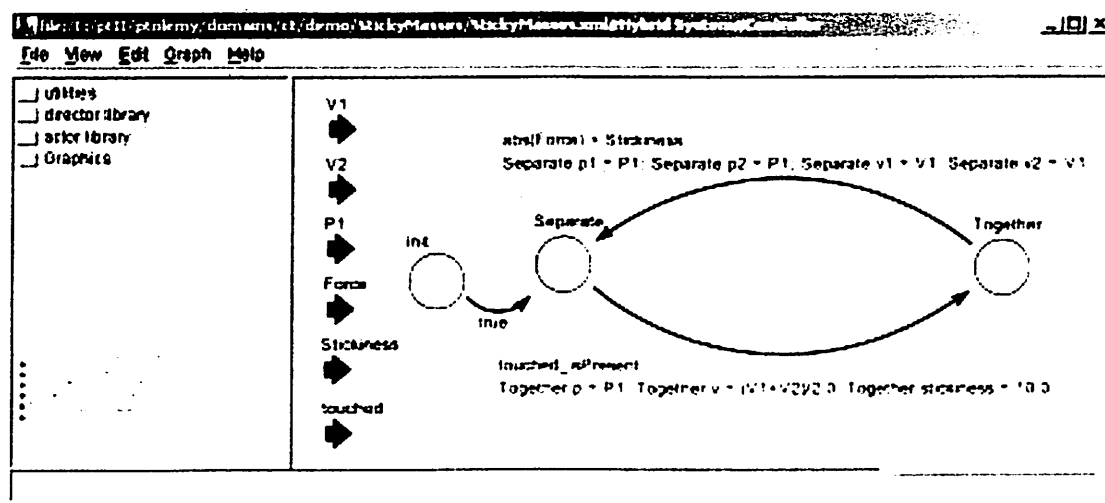

FIGURE 3. Visual rendition of a Ptolemy II model as a state transition diagram in Vergil (in the FSM domain).

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.

6. Ptolemy II Architecture

Ptolemy II 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.

Ptolemy II is modular, with a careful package structure that supports a layered approach. The *core packages* support the data model, or *abstract syntax*, of Ptolemy II designs. They also provide the *abstract semantics* that allows domains to interoperate with maximum information hiding. The *UI packages* provide support for our XML file format, called MoML, and a visual interface for constructing models graphically. The *library packages* provide actor libraries that are *domain polymorphic*, meaning that they can operate in a variety of domains. And finally, the *domain packages* provide domains, each of which implements a model of computation, and some of which provide their own, domain-specific actor libraries.

6.1 Core Packages

The core packages are shown in figure 4. This is a UML package diagram. 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* 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 that is extended in domains to control the execution of a model.
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, or where there is static analysis of the topology of a model associated with scheduling.
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. The key class is the Token class, which defines a set of polymorphic methods for operating on tokens, such as add(), subtract(), etc.

kernel

ComponentEntity
ComponentPort
ComponentRelation
CompositeEntity
Entity
Port
Relation

kernel.util

Attribute
ChangeListener
ChangeRequest
Configurable
ConfigurableAttribute
CrossRefList
DebugEvent
DebugListener
Debuggable
IllegalActionException
InternalErrorException
InvalidStateException
KernelException
KernelRuntimeException
NameDuplicationException
Nameable
NamedList
NamedObj
NoSuchItemException
PtolemyThread
RecorderListener
Settable
SingletonAttribute
SingletonConfigurableAttribute
StreamChangeListener
StreamListener
StringAttribute
StringUtilities
TransientSingletonConfigurableAttribute
ValueListener
Workspace

graph

CPO
DirectedAcyclicGraph
DirectedGraph
Graph
Inequality
InequalitySolver
InequalityTerm

math

ArrayStringFormat
Complex
ComplexArrayMath
ComplexBinaryOperation
ComplexMatrixMath
ComplexUnaryOperation
DoubleArrayMath
DoubleArrayStat
DoubleBinaryOperation
DoubleMatrixMath
DoubleUnaryOperation
ExtendedMath
FixPoint
FloatArrayMath
FloatBinaryOperation
FloatMatrixMath
FloatUnaryOperation
Fraction
IntegerArrayMath
IntegerBinaryOperation
IntegerMatrixMath
IntegerUnaryOperation
Interpolation
LongArrayMath
LongBinaryOperation
LongMatrixMath
LongUnaryOperation
Precision
Quantizer
SignalProcessing

actor

AbstractReceiver
Actor
AtomicActor
CompositeActor
Director
Executable
ExecutionListener
FiringEvent
IOPort
IORelation
Mailbox
Manager
NoRoomException
NoTokenException
QueueReceiver
Receiver
StreamExecutionListener
TypeAttribute
TypeConflictException
TypedActor
TypedAtomicActor
TypedCompositeActor
TypedIOPort
TypedIORelation
TypeEvent
TypeListener

actor.util

CQComparator
CalendarQueue
DoubleCQComparator
FIFOQueue
TimedEvent

actor.process

BoundaryDetector
Branch
BranchController
CompositeProcessDirector
MailboxBoundaryReceiver
NotifyThread
ProcessDirector
ProcessReceiver
ProcessThread
TerminateProcessException

actor.sched

Firing
NotSchedulableException
Schedule
ScheduleElement
Scheduler
StaticSchedulingDirector

data

ArrayToken
BooleanMatrixToken
BooleanToken
ComplexMatrixToken
ComplexToken
DoubleMatrixToken
DoubleToken
FixMatrixToken
FixToken
IntMatrixToken
IntToken
LongMatrixToken
LongToken
MatrixLowerBound
MatrixToken
MatrixUpperBound
Numerical
ObjectToken
RecordToken
ScalarToken
StringToken
Token

data.type

ArrayType
BaseType
RecordType
StructuredType
Type
TypeConstant
TypeLattice
Typeable

data.expr

ASCII_CharStream
ASTPtArrayConstructNode
ASTPtBitwiseNode
ASTPtFunctionNode
ASTPtFunctionalIfNode
ASTPtLeafNode
ASTPtLogicalNode
ASTPtMatrixConstructNode
ASTPtMethodCallNode
ASTPtProductNode
ASTPtRecordConstructNode
ASTPtRelationalNode
ASTPtRootNode
ASTPtSumNode
ASTPtUnaryNode
FixPointFunctions
MatrixParser
MatrixParserConstants
MatrixParserTokenManager
MatrixParserTreeConstants
Node
Parameter
ParseException
PtParser
PtParserConstants
PtParserTokenManager
PtParserTreeConstants
SimpleNode
Token
TokenMgrError
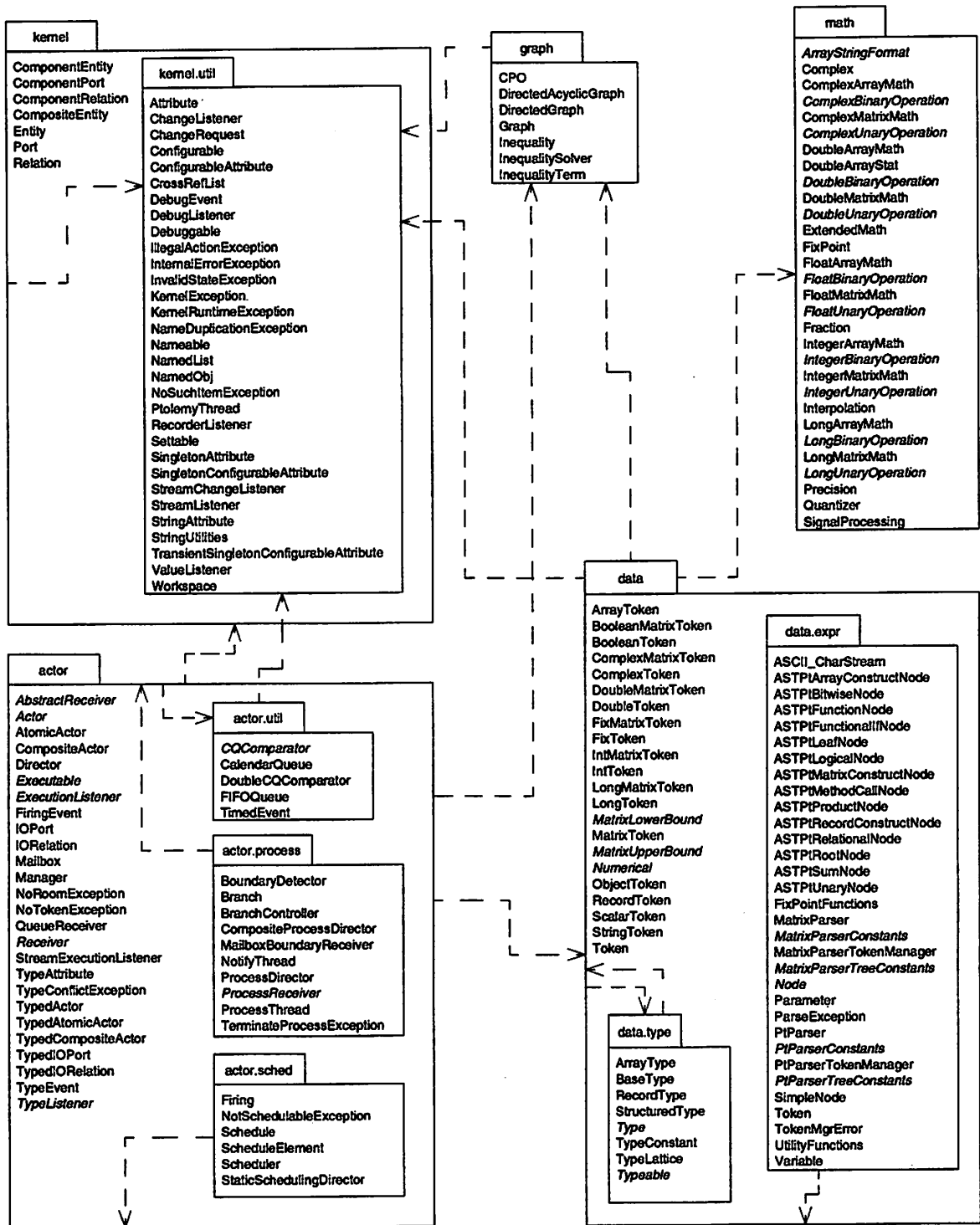UtilityFunctions
Variable

FIGURE 4. The core packages shown here support the data model, or abstract syntax, of Ptolemy II designs. They also provide the abstract semantics that allows domains to interoperate with maximum information hiding.

data.expr	This class supports an extensible expression language and an interpreter for that language. Parameters can have values specified by expressions. These expressions 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.
data.type	This package contains classes and interfaces for the type system.
graph	This package provides algorithms for manipulating and analyzing mathematical graphs. This package is expected to supply a growing library of algorithms. These algorithms support scheduling and analysis of Ptolemy II models.
kernel	This package provides the software architecture for the Ptolemy II data model, or *abstract syntax*. This abstract syntax has the structure of 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 composite entity.
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, a class supporting fractions, and a set of classes supporting fixed-point numbers.

6.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 5. This is a UML static structure diagram (see appendix A of this chapter). The key syntactic elements are boxes, which represent classes, the hollow arrow, which indicates generalization (or subclassing), and other lines, which indicate associations. Some lines have a small diamond, which indicates aggregation. The details of these classes will be discussed in subsequent chapters.

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 are 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, explained in the actors chapter, 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. The Executable and Actor interfaces are key to the Ptolemy II abstract semantics.

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 execu-

tion (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.

6.3 Domains

The domains in Ptolemy II are subpackages of the ptolemy.domains package, as shown in figure 6. These packages generally contain a kernel subpackage, which defines classes that extend those in the actor or kernel packages of Ptolemy II. The lib subpackage, when it exists, includes domain-specific actors.
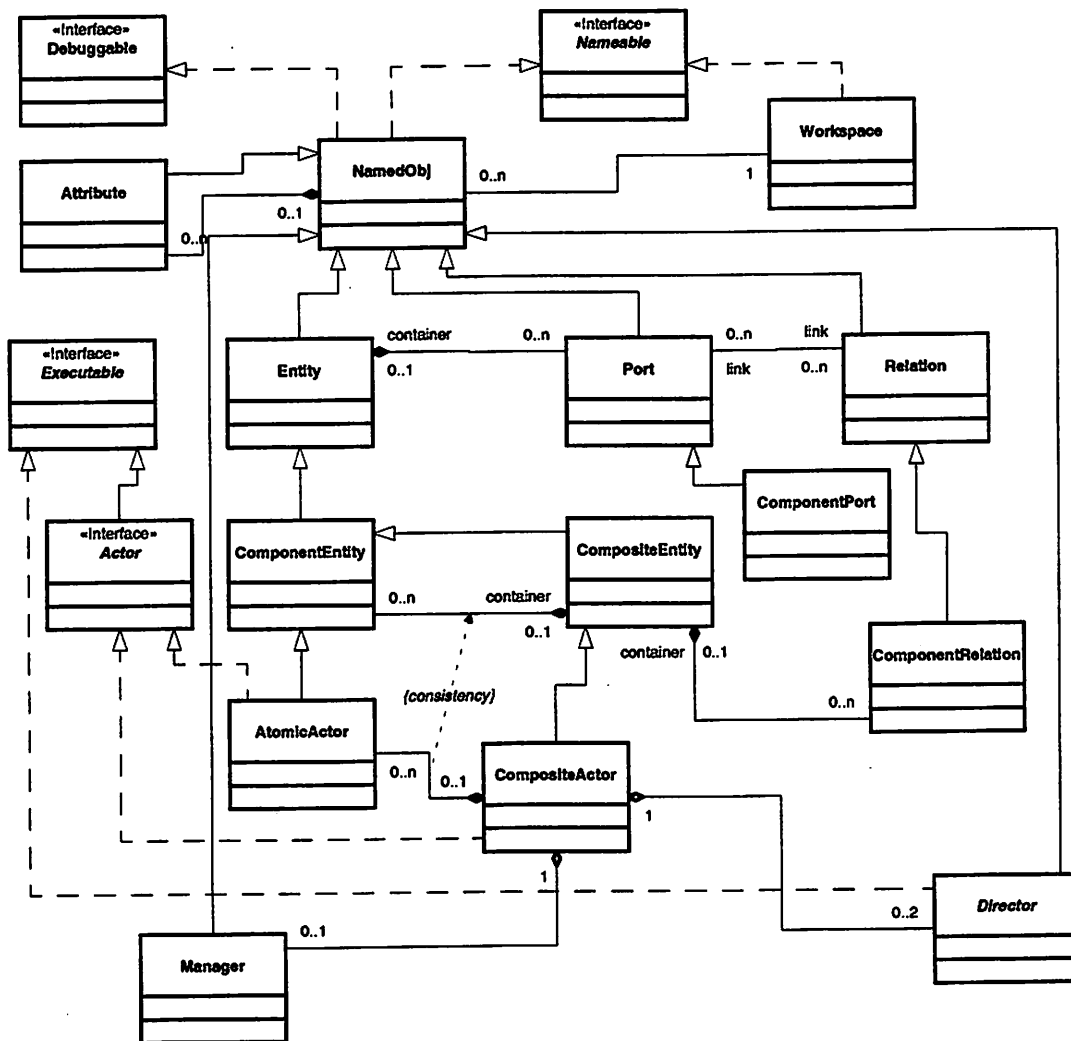

FIGURE 5. Some of the key classes in Ptolemy II. These are defined in the *kernel, kernel.util,* and *actor* packages. They define the Ptolemy II abstract syntax and abstract semantics.
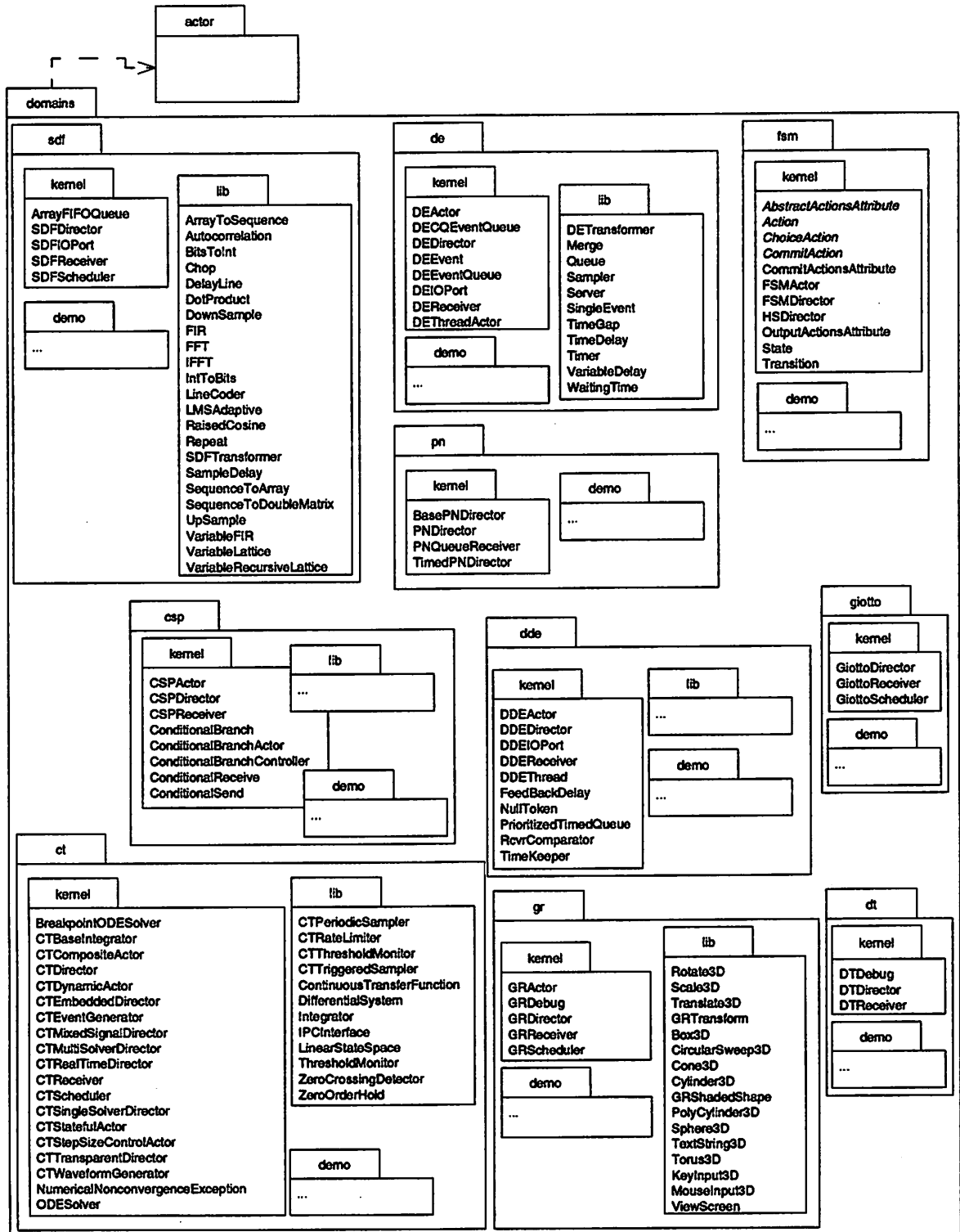
actor

domains

sdf

kernel
- ArrayFIFOQueue
- SDFDirector
- SDFIOPort
- SDFReceiver
- SDFScheduler

lib
- ArrayToSequence
- Autocorrelation
- BitsToInt
- Chop
- DelayLine
- DotProduct
- DownSample
- FIR
- FFT
- IFFT
- IntToBits
- LineCoder
- LMSAdaptive
- RaisedCosine
- Repeat
- SDFTransformer
- SampleDelay
- SequenceToArray
- SequenceToDoubleMatrix
- UpSample
- VariableFIR
- VariableLattice
- VariableRecursiveLattice

demo
...

de

kernel
- DEActor
- DECQEventQueue
- DEDirector
- DEEvent
- DEEventQueue
- DEIOPort
- DEReceiver
- DEThreadActor

lib
- DETransformer
- Merge
- Queue
- Sampler
- Server
- SingleEvent
- TimeGap
- TimeDelay
- Timer
- VariableDelay
- WaitingTime

demo
...

pn

kernel
- BasePNDirector
- PNDirector
- PNQueueReceiver
- TimedPNDirector

demo
...

fsm

kernel
- *AbstractActionsAttribute*
- *Action*
- *ChoiceAction*
- *CommitAction*
- CommitActionsAttribute
- FSMActor
- FSMDirector
- HSDirector
- OutputActionsAttribute
- State
- Transition

demo
...

csp

kernel
- CSPActor
- CSPDirector
- CSPReceiver
- ConditionalBranch
- ConditionalBranchActor
- ConditionalBranchController
- ConditionalReceive
- ConditionalSend

lib
...

demo
...

dde

kernel
- DDEActor
- DDEDirector
- DDEIOPort
- DDEReceiver
- DDEThread
- FeedBackDelay
- NullToken
- PrioritizedTimedQueue
- RcvrComparator
- TimeKeeper

lib
...

demo
...

giotto

kernel
- GiottoDirector
- GiottoReceiver
- GiottoScheduler

demo
...

ct

kernel
- BreakpointODESolver
- CTBaseIntegrator
- CTCompositeActor
- CTDirector
- CTDynamicActor
- CTEmbeddedDirector
- CTEventGenerator
- CTMixedSignalDirector
- CTMultiSolverDirector
- CTRealTimeDirector
- CTReceiver
- CTScheduler
- CTSingleSolverDirector
- CTStatefulActor
- CTStepSizeControlActor
- CTTransparentDirector
- CTWaveformGenerator
- NumericalNonconvergenceException
- ODESolver

lib
- CTPeriodicSampler
- CTRateLimiter
- CTThresholdMonitor
- CTTriggeredSampler
- ContinuousTransferFunction
- DifferentialSystem
- Integrator
- IPCInterface
- LinearStateSpace
- ThresholdMonitor
- ZeroCrossingDetector
- ZeroOrderHold

demo
...

gr

kernel
- GRActor
- GRDebug
- GRDirector
- GRReceiver
- GRScheduler

demo
...

lib
- Rotate3D
- Scale3D
- Translate3D
- GRTransform
- Box3D
- CircularSweep3D
- Cone3D
- Cylinder3D
- GRShadedShape
- PolyCylinder3D
- Sphere3D
- TextString3D
- Torus3D
- KeyInput3D
- MouseInput3D
- ViewScreen

dt

kernel
- DTDebug
- DTDirector
- DTReceiver

demo
...

FIGURE 6. Package structure of Ptolemy II domains.

6.4 Library Packages

Most domains extend classes in the actor package to give a specific semantic interpretation to an interconnection of actors. It is possible, and strongly encouraged, to define actors in such a way that they can operate in multiple domains. Such actors are said to be *domain polymorphic*. Actor that are domain polymorphic are organized in the packages shown in figure 7. These packages are briefly described below:

actor.lib   This subpackage is the main library of polymorphic actors. Eventually, this pack-

actor

actor.lib

AbsoluteValue
AddSubtract
ArrayAppend
ArrayElement
ArrayExtract
Average
Bernoulli
Clock
Commutator
Const
CurrentTime
DB
Discard
DiscreteRandomSource
Distributor
Expression
FileWriter
Gaussian
IIR
Interpolator
Lattice
LevinsonDurbin
Maximum
MaxIndex
Minimum
Multiplexor
MultiplyDivide
PhaseUnwrap
PoissonClock
Pulse
Quantizer
Ramp
RandomSource
Reader
RealTimeDelay
RecordAssembler
RecordDisassembler
Recorder
RecursiveLattice
Remainder
Scale
Select
Sequencer
SequenceActor
SequenceSource
SequentialClock
Sink
Sleep
Source
Switch
Test
TimedActor
TimedSource
Transformer
TrigFunction
Uniform
VariableClock
Writer

actor.lib.conversions

CartesianToComplex
CartesianToPolar
ComplexToCartesian
ComplexToPolar
DoubleToFix
FixToDouble
FixToFix
PolarToCartesian
PolarToComplex
Round

actor.lib.gui

BarGraph
Display
HistgramPlotter
MatrixViewer
MatrixVisualizer
Plotter
SequencePlotter
SequenceScope
SketchedSource
TimedPlotter
TimedScope
XYPlotter
XYScope

plot

actor.gui

actor.lib.javasound

AudioSink
AudioSource

actor.lib.logic

Comparator
Equals
LogicalNot
LogicFunction

FIGURE 7. Packages containing domain-polymorphic actors.

age will be reorganized into subpackages.

actor.lib.gui This subpackage is a library of polymorphic actors with user interface components, such as plotters.

actor.lib.conversions

This subpackage provides domain polymorphic actors that convert data between different types.

actor.lib.javasound

This package provides sound actors on systems that are running Java 1.3 or later.

actor.lib.logic This subpackage provides actors that perform logical functions like AND, OR and NOT.

6.5 User Interface Packages

The UI packages provide support for our XML file format, called MoML, and a visual interface for constructing models graphically, called Vergil. These packages are organized as shown in figure 6.1. The intent of each package is described below:

actor.gui This subpackage contains the configuration infrastructure, which supports modular construction of user interfaces that are themselves Ptolemy II models.

actor.gui.style This package contains classes that decorate attributes to serve as hints to a user interface about how to present these attributes to the user.

gui This package contains generically useful user interface components.

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

moml This package contains classes support our XML modeling markup language (MoML), which is used to describe Ptolemy II models.

plot This package and its subpackages provides two-dimensional signal plotting widgets.

vergil This package and its subpackages contains the Ptolemy II graphical user interface

7. Capabilities

Ptolemy II is a second generation system. Its predecessor, Ptolemy Classic, 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 [20]. Ptolemy II includes a number of domains that support design of concurrent systems at a much higher level of abstraction, at the level of their software architecture. Some of these domains use Java threads as an underlying mechanism, while others offer an alternative to Java threads that is much more efficient and scalable.

- *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
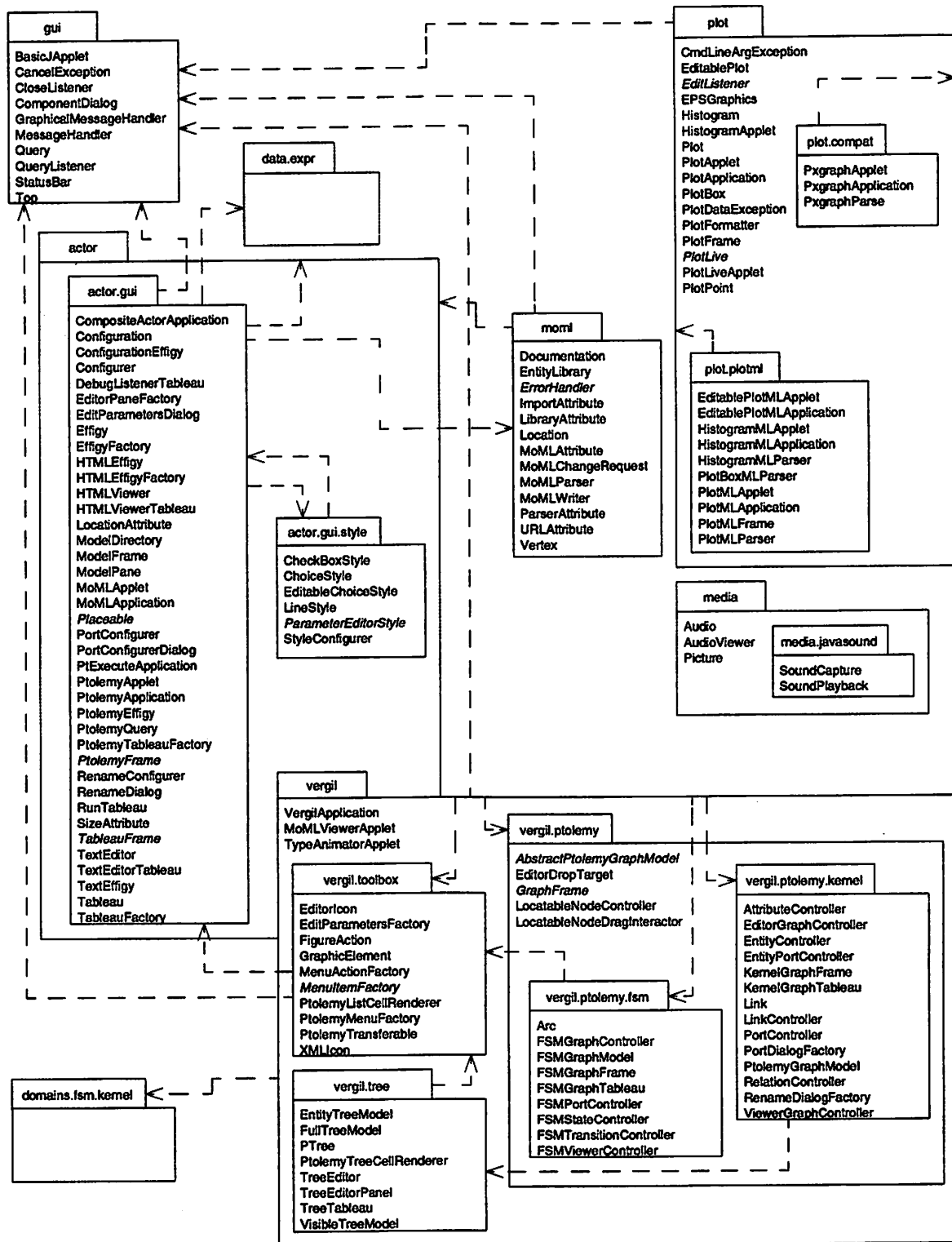
FIGURE 6.1. Packages in Ptolemy II that support user interfaces, including the MoML XML schema and the Vergil visual editor.

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 via a well-defined abstract semantics.* Ptolemy Classic 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 Ptolemy Classic. 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 [17] built upon the lower level synchronization primitives of Java.

- *A software architecture based on object modeling.* Since Ptolemy Classic was constructed, software engineering has seen the emergence of sophisticated object modeling [30][36][38] and design pattern [11] 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 [35].

- *A truly polymorphic type system.* Ptolemy Classic 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 [32]. The type system is described in [43].

- *Domain-polymorphic actors.* In Ptolemy Classic, 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*.

7.1 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, Java RMI, or DCOM, 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.

- *Code generation.* Ptolemy II has an evolving code generation mechanism that is very different from that in Ptolemy Classic. In Ptolemy Classic, each component has to have a definition in the target language, and the code generator merely stitches together these components. In Ptolemy II, components are defined in Java, and the Java definition is parsed. An API for performing optimization transformations on the abstract syntax tree is defined, and then compiler back ends can be used to generate target code. A preliminary implementation of this approach is described in [41] and [42].

- *Integrated verification tools.* Modern verification tools based on model checking [16] 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.

- *Reflection of dynamics.* Java supports reflection of static structure, but not of dynamic properties of process-based objects. For example, the data layout required to communicate with an object is available through the reflection package, but the communication protocol is not. We plan to extend the notion of reflection to reflect such dynamic properties of objects.

- *Meta modeling.* The domains in Ptolemy II are constructed based on an intuitive understanding of a useful class of modeling techniques, and then the support infrastructure for specifying and executing models in the domain are built by hand by writing Java code. Others have built tools that have the potential of improving on this situation by *meta modeling*. In Dome (from Honeywell) and GME (from Vanderbilt), for example, a modeling strategy itself is modeled, and user interfaces supporting that modeling strategy are synthesized from that model. We can view the current component-based architecture of Vergil as a starting point in this direction. In the future, we expect to see much more use of Ptolemy II itself to define and construct Ptolemy II domains and their user interfaces.

8. References

[1] R. Allen and D. Garlan, "Formalizing Architectural Connection," in *Proc. of the 16th International Conference on Software Engineering* (ICSE 94), May 1994, pp. 71-80, IEEE Computer Society Press.

[2] 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.

[3] 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.

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

[5] Randy Brown, "CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem", Communications of the ACM, October 1998, Volume 31, Number 10.

[6] 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).

[7] 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.

[8] John Davis II, "Order and Containment in Concurrent System Design," Ph.D. thesis, Memorandum UCB/ERL M00/47, Electronics Research Laboratory, University of California, Berkeley, September 8, 2000.

[9] S. A. Edwards, "The Specification and Execution of Heterogeneous Synchronous Reactive Systems," **Ph.D. thesis**, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31. (http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/)

[10] C. Fong, "Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II," Master's Report, Memorandum UCB/ERL M01/9, Electronics Research Laboratory, University of California, Berkeley, January 2001.

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

[12] 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)

[13] M. Goel, *Process Networks in Ptolemy II*, MS Report, ERL Technical Report UCB/ERL No. M98/69, University of California, Berkeley, CA 94720, December 16, 1998.

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

[15] 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.

[16] 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.

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

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

[19] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

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

[21] Edward A. Lee, "What's Ahead for Embedded Software?," IEEE Computer, September 2000, pp. 18-26.

[22] 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)

[23] 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/)

[24] Edward A. Lee and Yuhong Xiong, "System-Level Types for Component-Based Design, "Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University o f California, Berkeley, CA 94720, USA, February 29, 2000.

[25] J. Liu, *Continuous Time and Mixed-Signal Simulation in Ptolemy II*, MS Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998.

[26] Jie Liu and Edward A. Lee, "Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics," *Proc. of the 2000 IEEE International Conference on Control Applications and IEEE Symposium on Computer-Aided Control System Design* (CCA/CACSD'00), Anchorage, AK, September 25-27, 2000. pp. 95-100

[27] D. C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.

[28] 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.

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

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

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

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

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

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

[35] J. Reekie, S. Neuendorffer, C. Hylands and E. A. Lee, "Software Practice in the Ptolemy Project," Technical Report Series, GSRC-TR-1999-01, Gigascale Silicon Research Center, University of California, Berkeley, CA 94720, April 1999.

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

[37] J. Rowson and A. Sangiovanni-Vincentelli, "Interface Based Design," *Proc. of DAC '97*.

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

[39] S. Saracco, J. R. W. Smith, and R. Reed, *Telecommunications Systems Engineering Using SDL*, North-Holland - Elsevier, 1989.

[40] N. Smyth, *Communicating Sequential Processes Domain in Ptolemy II*, MS Report, UCB/ERL Memorandum M98/70, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998.

[41] J. Tsay, "A Code Generation Framework for Ptolemy II," ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/codegen).

[42] Jeff Tsay, Christopher Hylands and Edward Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[43] Yuhong Xiong and Edward A. Lee, "An Extensible Type System for Component-Based Design," *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000 . LNCS 1785.