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

OVERVIEW OF THE PTOLEMY PROJECT

JULY 2, 2003

Technical Memorandum UCB/ERL M03/25
http://ptolemy.eecs.berkeley.edu/

Christopher Hylands
Edward Lee
Jie Liu
Xiaojun Liu
Stephen Neuendorffer
Yuhong Xiong
Yang Zhao
Haiyang Zheng

1. Background

The Ptolemy Project is an informal group of researchers that is part of Chess (the center for hybrid and embedded software systems) at U.C. Berkeley; see "Acknowledgements" on page 28 for a list participants. This projects conducts foundational and applied research in software based design techniques for embedded systems. Ptolemy II is the current software infrastructure of the Ptolemy Project. For the participants in the Ptolemy Project, Ptolemy II is first and foremost a laboratory for experimenting with design techniques. It is published freely in open-source form. Distribution of open-source software complements more traditional publication media, and serves as a clear, unambiguous, and complete description our research results. Also, the open architecture and open source encourages researchers to build their own methods, leveraging and extending the core infrastructure provided by the software. This creates a community where much of the dialog is through the software. In addition, the freely available software encourages designers to try out the new design techniques that are introduced and give feedback to the Ptolemy Project. This helps guide further research. Finally, the open source software encourages commercial providers of software tools to commercialize the research results, which then helps to maximize the impact of the work.

Ptolemy II is the third generation of design software to emerge from this group, with each generation bringing a new set of problems being addressed, new emphasis, and (largely) a new group of contributors.

1.1 Gabriel (1986-1991)

The first generation of software created by this was group was called Gabriel [16]. It was written in Lisp and aimed at signal processing. It was during the construction of Gabriel that synchronous dataflow (SDF) block diagrams and both sequential and parallel scheduling techniques for SDF models matured. Gabriel included code generators for programmable DSPs that produced efficient assembly code for certain processors (notably, Motorola processors). Gabriel included hardware/software co-simulators, where parallel code generators would produce assembly code which then ran on instruction set simulators within a hardware simulation of a multiprocessor architecture. Gabriel had a graphical user interface built on top of Vem, written by Rick Spickelmeyer, which was originally designed for schematic capture in VLSI CAD. It used Oct, which was the design database developed by the Berkeley CAD group under the leadership of Professor Richard Newton.

1.2 Ptolemy Classic (1990-1997)

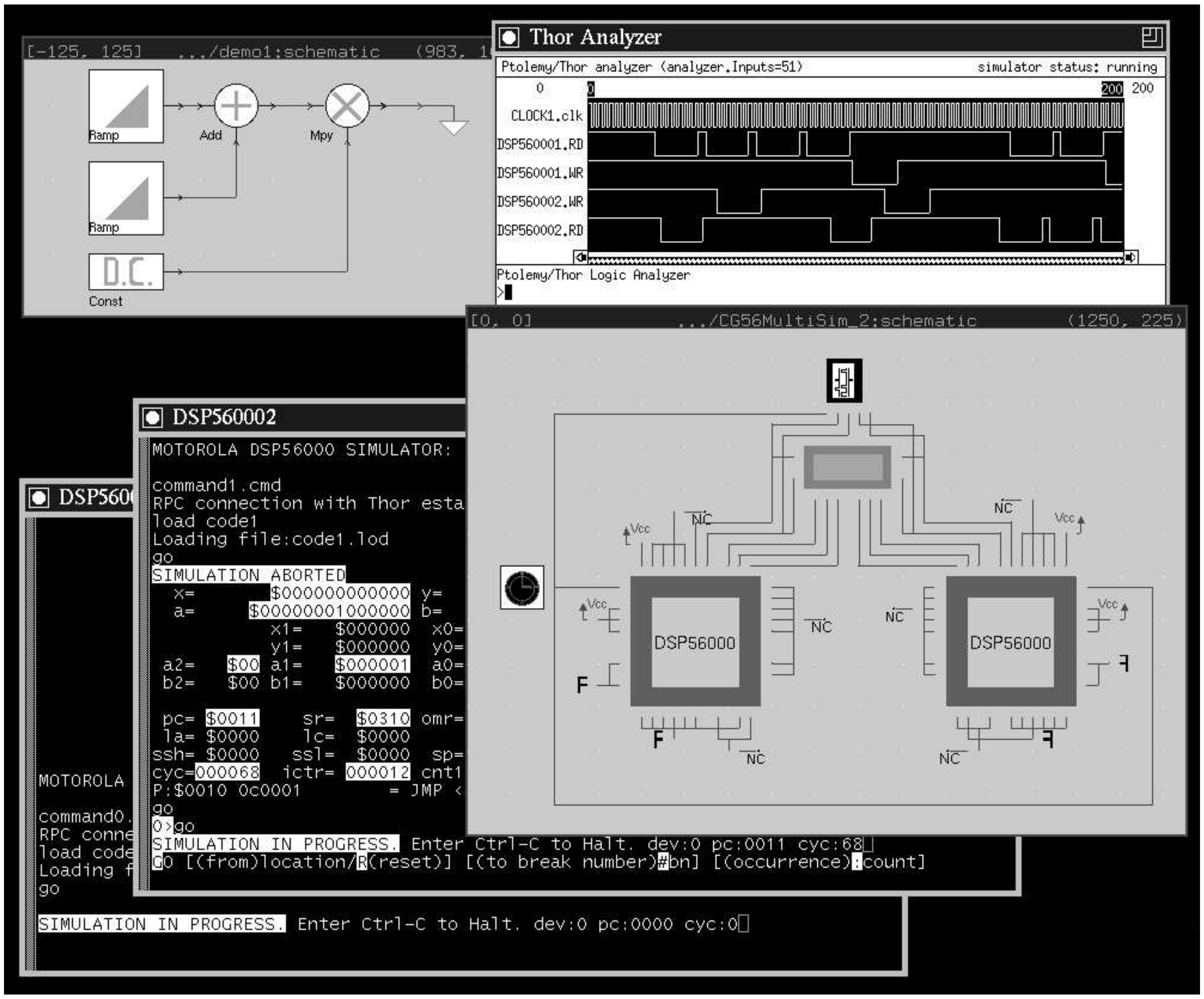
Ptolemy Classic, started jointly by Professors Edward Lee and Dave Messerschmitt in 1990, was written in C++ [19]. It was the first modeling environment to systematically support multiple models of computation, hierarchically combined. We ported the SDF capabilities from Gabriel, and also developed boolean dataflow (BDF), dynamic dataflow (DDF), multidimensional synchronous dataflow (MDSDF) and process networks (PN) domains. We also ported the DSP code generators, and created C and VHDL code generators as well. We developed the concept of “targets,” which encapsulated knowledge about specific hardware platforms, and demonstrated construction of models that executed on attached embedded processors (such as S-bus cards with DSPs), including models that executed jointly on a Unix host and the attached embedded processor. We developed a discrete-event domain and demonstrated joint modeling of communication networks and signal processing, and also developed a hardware simulation domain called Thor, which was adapted from an open-source hardware simulator by the same name (see figure 1). We made major contributions to SDF scheduling techniques, including introducing the concept of “single appearance schedules” (which minimize generated code size and enable extensive use of inlining of generated code). We also introduced “higher-order components,” which greatly increased the expressiveness of visual syntaxes [65]. The Ptolemy Classic user interface was an extension of the Gabriel interface, still based on Oct and Vem, but extended by Tycho (written in Itcl, an object-oriented extension of Tcl/Tk). Portions of Ptolemy Classic were commercialized as part of the Agilent ADS system, and methods from Ptolemy Classic were used in Cadence’s SPW system.

1.3 Ptolemy II (1996-?)

The Ptolemy Project (as it was now known) began working on Ptolemy II in 1996. The major reasons for starting over were to exploit the network integration, migrating code, built-in threading, and user-interface capabilities of Java. Ptolemy II introduced the notion of domain polymorphism (where components could be designed to be able to operate in multiple domains) and modal models (where finite state machines are combined hierarchically with other models of computation). We built a continuous-time domain, which combined with the modal modeling capability, yields hybrid system modeling. Ptolemy II has a sophisticated type system with type inference and data polymorphism (where components can be designed to operate on multiple data types), and a rich expression language. The concept of behavioral types emerged (where components and domains could have interface definitions that describe not just static structure, as with traditional type systems, but also dynamic behavior) [68]. Some (but not all) of the SDF capabilities from Ptolemy Classic were ported, and the heterochronous dataflow model was introduced [38]. We contributed to a user-interface toolkit (called Diva) based on Java, built a user interface for Ptolemy II (called Vergil) based on Diva, designed a Java plotter (PtPlot), and introduced a 3-D animation domain. We built models that could be used as applets in a web browser. And we built numerous experimental domains that explored real-time and distributed computing (distributed discrete events (DDE), timed multitasking (TM), Giotto, and component interaction (CI)). As for code generation, the tactic in Ptolemy II is significantly different than that in Gabriel or Ptolemy Classic. Instead of components as generators, Ptolemy II uses a component-specialization framework built on top of a Java compiler toolkit called Soot. Ptolemy II uses XML for its persistent data representation, and has introduced the concept of migrating models [98].

2. Modeling and Design

The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent sys-


[-125, 125]   .../demo1:schematic   (983, 1
Ramp
Add
Mpy
Ramp
D.C.
Const
Thor Analyzer
Ptolemy/Thor analyzer (analyzer.Inputs=51)
simulator status: running
0
0
200
200
CLOCK1.clk
DSP560001.RD
DSP560001.WR
DSP560002.WR
DSP560002.RD
Ptolemy/Thor Logic Analyzer
[0, 0]   .../CG56MultiSim_2:schematic   (1250, 225)
Vcc
NC
NC
Vcc
Vcc
DSP56000
NC
NC
DSP56000
Vcc
F
F
NC
NC
DSP560002
DSP560

```
MOTOROLA DSP56000 SIMULATOR:

command1.cmd
RPC connection with Thor esta
load code1
Loading file:code1.lod
go
SIMULATION ABORTED
  x=         $000000000000 y=
  a=       $00000001000000 b=
              x1=   $000000  x0=
              y1=   $000000  y0=
 a2=   $00 a1=   $000001  a0=
 b2=   $00 b1=   $000000  b0=

 pc= $0011    sr=  $0310 omr=
 la= $0000    lc=  $0000
ssh= $0000   ssl=  $0000  sp=
cyc=000068  ictr= 000012 cnt1
P:$0010 0c0001        = JMP <
go
0>go
SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0011 cyc:68
GO [(from)location/R(reset)] [(to break number)#bn] [(occurrence):count]
```

```
MOTOROLA

command0.
RPC conne
load code
Loading f
go

SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0000 cyc:0
```

FIGURE 1. Ptolemy Classic screen image (from 1993) showing an SDF graph at the upper left that is automatically mapped and scheduled onto the two processor architecture, whose model is at the lower right (in the "Thor" domain). Assembly code for the two processor is generated, and then ISA simulators of the processors (provided by Motorola, lower left) interact with the Thor-domain simulation, resulting in the logic analyzer trace at the upper right.

tems. The focus is on *embedded systems* [66], 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 networking, signal processing, feedback control, mode changes, 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.

2.1 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 key feature of embedded software is that it engages the physical world, and hence has temporal constraints that desktop software does not share.

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 components.

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

2.2 Actor-Oriented Design

Most (but not all) models of computation in Ptolemy II support *actor-oriented design*. This contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components. Components called actors execute and communicate with other actors in a model, as illustrated in figure 2. Like objects, actors have a well defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes ports that represent points of communication for an actor, and parameters which are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed, but not always. The "port/parameters" shown in figure 2 function as both ports and parameters.

Central to actor-oriented design are the communication channels that pass data from one port to another according to some messaging scheme. Whereas with object-oriented design, components interact primarily by transferring control through method calls, in actor-oriented design, they interact by sending messages through channels. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

Like actors, a model may also define an external interface; this interface is called its *hierarchical abstraction*. This interface consists of external ports and external parameters, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that compose the model. External parameters of a model can be used to determine the values of the parameters of actors inside the model.

Taken together, the concepts of models, actors, ports, parameters and channels describe the *abstract syntax* of actor-oriented design. This syntax can be represented concretely in several ways, such as graphically, as in figure 4, in XML as in figure 3, or in a program designed to a specific API (as in SystemC). Ptolemy II offers all three alternatives.

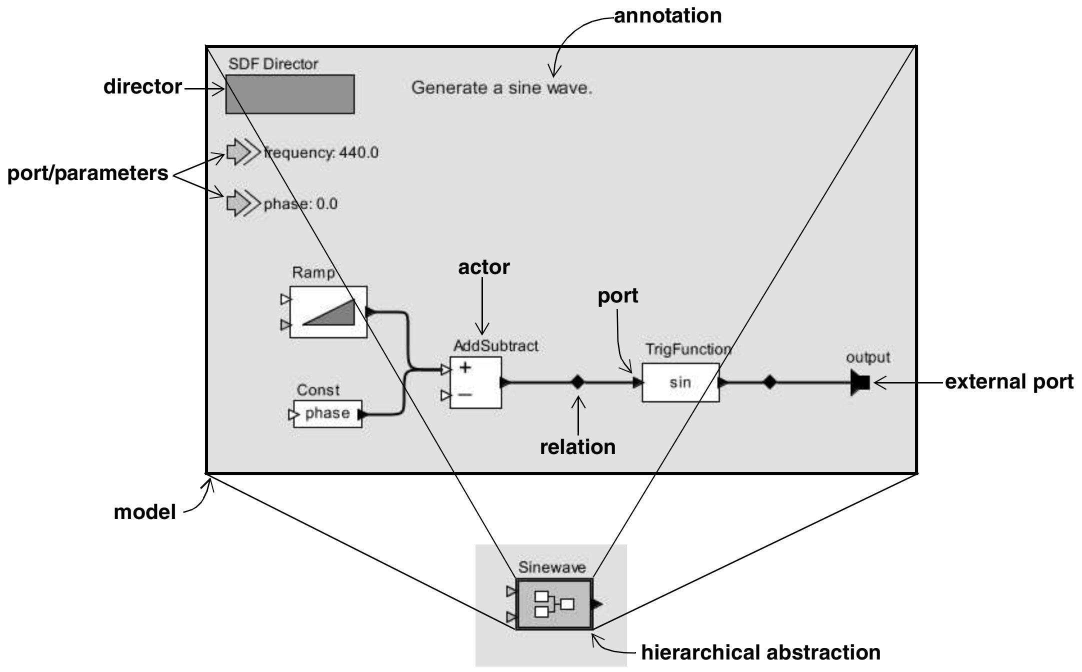


FIGURE 2. Illustration of an actor-oriented model (above) and its hierarchical abstraction (below).

```
<class name="Sinewave">
   <property name="samplingFrequency" value="8000.0"/>
   <property name="frequency" value="440.0"/>
   <property name="phase" value="0.0"/>
   <property name="SDF Director" class="ptolemy.domains.sdf.kernel.SDFDirector"/>
   <port name="output"><property name="output"/>
   <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
      <property name="init" value="phase"/>
      <property name="step" value="frequency*2*PI/samplingFrequency"/>
   </entity>
   <entity name="TrigFunction" class="ptolemy.actor.lib.TrigFunction">
      <property name="function" value="sin" class="ptolemy.kernel.util.StringAttribute"/>
   </entity>
   <relation name="relation"/>
   <relation name="relation2"/>
   <link port="output" relation="relation2"/>
   <link port="Ramp.output" relation="relation"/>
   <link port="TrigFunction.input" relation="relation"/>
   <link port="TrigFunction.output" relation="relation2"/>
</class>
```

FIGURE 3. An XML representation of a simplified sinewave source.

It is important to realize that the syntactic structure of an actor-oriented design says little about the semantics. The semantics is largely orthogonal to the syntax, and is determined by a model of computation. The model of computation might give operational rules for executing a model. These rules determine when actors perform internal computation, update their internal state, and perform external communication. The model of computation also defines the nature of communication between components.

Our notion of actor-oriented modeling is related to the work of Gul Agha and others. The term actor was introduced in the 1970's by Carl Hewitt of MIT to describe the concept of autonomous reasoning agents [48]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1-5]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. We have further developed the term to embrace a larger family of models of concurrency that are often more constrained than general message passing. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Moreover, although communication is still through some form of message passing, it need not be strictly asynchronous.

Actor-oriented modeling has been around for some time, and is in widespread use through such programs as Simulink, from The Mathworks, LabView, from National Instruments, and many others. It is gaining broader legitimacy through the efforts of OMG in UML-2 [101], for example, some of which has its roots in the actor-oriented framework ROOM [116]. Many research projects are based on some form of actor-oriented modeling, but the Ptolemy Project is unique in the breadth of exploration of semantic alternatives and in the commitment made to a particular model of computation within a domain.

2.3 Architecture Design

Architecture description languages (ADLs), such as Wright [6] and Rapide [85], 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 [6], 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 [6], we use a technique much more powerful than type checking alone, namely polymorphism [68].

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 embed 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 [6], we have developed a more abstract formal framework that describes models of computation at a meta level [71]. 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 [6].

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. The utility of a model of computation stems from the modeling properties that apply to all similar models. For many models of computation these properties are derived through formal mathematics. Depending on the model of computation, the model may be determinate [55], statically schedulable [72], or time safe [47]. Because of its modeling properties, a model of computation represents a style of modeling that is useful in any circumstance where those properties are desirable. In other words, models of computation form design patterns of component interaction, in the same sense that Gamma, et al. describe design patterns in object oriented languages [35].

For a particular application, an appropriate model of computation does not impose unnecessary constraints, and at the same time is constrained enough to result in useful derived properties. For example, by restricting the design space to synchronous designs, Scenic [74] enables cycle-driven simulation [41], which greatly improves execution efficiency over more general discrete-event models of computation (such as that found in VHDL). However, for applications with multirate behavior, synchronous design can be constraining. In such cases, a less constrained model of computation, such as synchronous dataflow [72] or Kahn process networks [55] may be more appropriate. One drawback of this relaxation of synchronous design constraints is that buffering becomes more difficult to analyze. On the other hand, techniques exist for synchronous dataflow that allow co-optimization of memory usage and execution latency [118] that would otherwise be difficult to apply to a multirate system. Selecting an appropriate model of computation for a particular application is often difficult, but this is a problem we should embrace instead of avoiding.

In this section, we describe models of computation that are implemented in Ptolemy II domains. 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 4.

Ptolemy II models are (clustered, or hierarchical) graphs of the form of figure 4, 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 Component Interaction - CI

The component interaction (CI) domain, created by Xiaojun Liu and Yang Zhao, models systems that blend data-driven and demand-driven styles of computation. As an example, the interaction between a web server and a browser is mostly demand-driven. When the user clicks on a link in the browser, it pulls the corresponding page from the web server. A stock-quote service can use a data-driven style of computation. The server generates events when stock prices change. The data drive the clients to update their displayed information. Such push/pull interaction between a data producer and consumer is common in distributed systems, and has been included in middleware services, most notably in the CORBA event service. These services motivated the design of this domain to study the interaction models in distributed systems, such as stock-quote services, traffic or weather information systems. Other applications include database systems, file systems, and the Click modular router [57].

An actor in a CI model can be active, which means it possesses its own thread of execution. For example, an interrupt source of an embedded system can be modeled as an active source actor. Such a source generates events asynchronously with respect to the software execution on the embedded processor. CI models can be used to simulate and study how the embedded software handles the asynchronous events, such as external interrupts and asynchronous I/O.

3.2 Communicating Sequential Processes - CSP

In the CSP domain (communicating sequential processes), created by Neil Smyth [117], 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) [51] and Milner’s *calculus of communicating systems* (CCS) [90]. 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

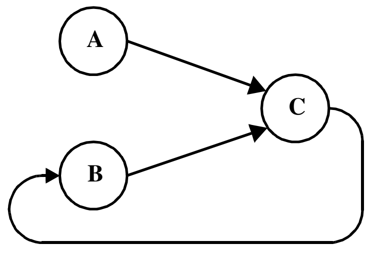


FIGURE 4. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

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.

3.3 Continuous Time - CT

In the CT domain (continuous time), created Jie Liu [81], 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.

Mixed Signal Models. 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* [82]. 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.

Modal Models and Hybrid Systems. 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. Such modal models are often called *hybrid systems*.

3.4 Discrete-Events - DE

In the discrete-event (DE) domain, created by Lukito Muliadi [94], 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 [17] 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.

3.5 Distributed Discrete Events - DDE

The distributed discrete-event (DDE) domain, created by John Davis [26], 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.

3.6 Discrete Time - DT

The discrete-time (DT) domain, written by Chamberlain Fong [32], 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.

3.7 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 expressing control logic and for building modal models (models with distinct modes of operation, where behavior is different in each mode). FSM models are amenable to in-depth formal analysis, and thus can be used to avoid surprising behavior.

**Charts.* 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 [42]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [45].

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 [38]. 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 Statecharts. When combined with process networks, they resemble SDL [115].

3.8 Process Networks - PN

In the process networks (PN) domain, created by Mudit Goel [39], 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 [55].

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 [56], 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.

3.9 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 [70]). 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.

3.10 Giotto

The Giotto domain, created by Christoph Meyr Kirsch, realizes a model of computation developed by Tom Henzinger, Christoph Kirsch, Ben Horowitz and Haiyang Zheng [44]. This domain has a time-triggered flavor, where each actor is invoked periodically with a specified period. The domain is designed to work with the FSM domain to realize modal models. It is intended for hard-real-time systems, where resource allocation is precomputed.

3.11 Synchronous/Reactive - SR

In the synchronous/reactive (SR) domain, written by Paul Whitaker [121] implements a model of computation [11] where 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 [13], Signal [12], Lustre [23], and Argos [86].

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. The SR domain implementation in Ptolemy II is similar to the SR implementation in Ptolemy Classic by Stephen Edwards[28].

3.12 Timed Multitasking - TM

The timed multitasking (TM) domain, created by Jie Liu [80], supports the design of concurrent real-time software. It assumes an underlying priority-driven preemptive scheduler, such as that typically found in a real-time operating systems (RTOS). But the behavior of models is more deterministic than that obtained by more ad hoc uses of an RTOS.

In TM, each actor executes (conceptually) as a concurrent task. It is a timed domain, meaning that there is a notion of "model time" that advances monotonically and uniformly. Each actor has a specified execution time T, and it delays the production of the outputs until it has had access to the CPU for that specified amount of time (in model time, which may or may not match real time). Actors execute when they receive new inputs, so the execution is event driven. Conceptually, the actor begins execution at some time t, and its output is produced at time $t + T + P$, where T is the declared execution time, and P is the amount of time where the actor is suspended due to being preempted by a higher priority actor. At any given model time t, the task with the highest priority that has received inputs but not yet produced its outputs has the CPU. All other tasks are suspended.

TM offers a way to design real-time systems that is more deterministic than ad hoc uses of an RTOS. In particular, typically, a task produces outputs at a time that depends on the actual execution time of the task, rather than on some declared parameter. This means that consumers of that data may or may not see updates to the data, depending on when their execution occurs relative to the actual execution time. Thus, the computational results that are produced depend on the actual execution time.

TM avoids this by declaring the time that elapses before production of the outputs. By maintaining model time correctly, TM ensures that the data computation is deterministic, irrespective of actual execution time.

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 [71].

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 [6] and Metropolis (http://www.gigascale.org/metropolis/), for example. 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. Wright, for example, uses rendezvous, which 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 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.*

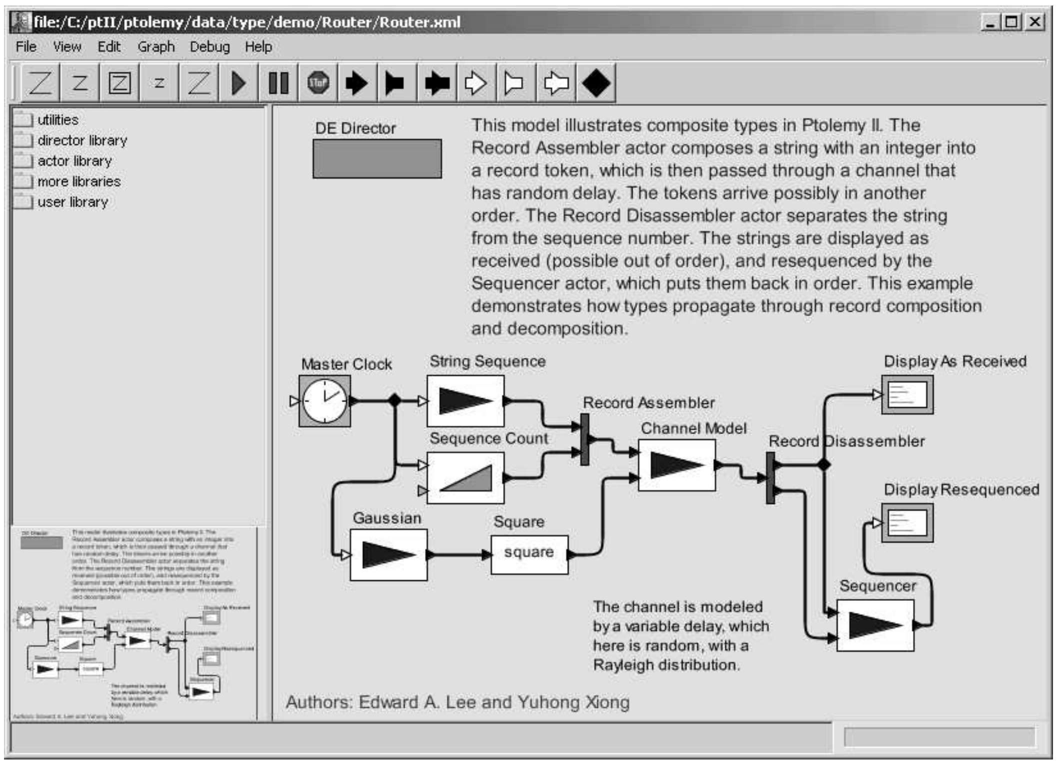
1. Consider the difference between the telephone (rendezvous) and email (asynchronous message passing). If you are trying to schedule a meeting between four busy people, getting them all on a conference call would lead to a quick resolution of the meeting schedule. Scheduling the meeting by email could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

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 5 and 6 show two different visual renditions of Ptolemy II models. Both renditions are constructed in Vergil, the visual editor framework in Ptolemy II designed by Steve Neuendorffer. In figure 5, 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 6 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 textual 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 5 and 6 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



This model illustrates composite types in Ptolemy II. The Record Assembler actor composes a string with an integer into a record token, which is then passed through a channel that has random delay. The tokens arrive possibly in another order. The Record Disassembler actor separates the string from the sequence number. The strings are displayed as received (possible out of order), and resequenced by the Sequencer actor, which puts them back in order. This example demonstrates how types propagate through record composition and decomposition.

The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

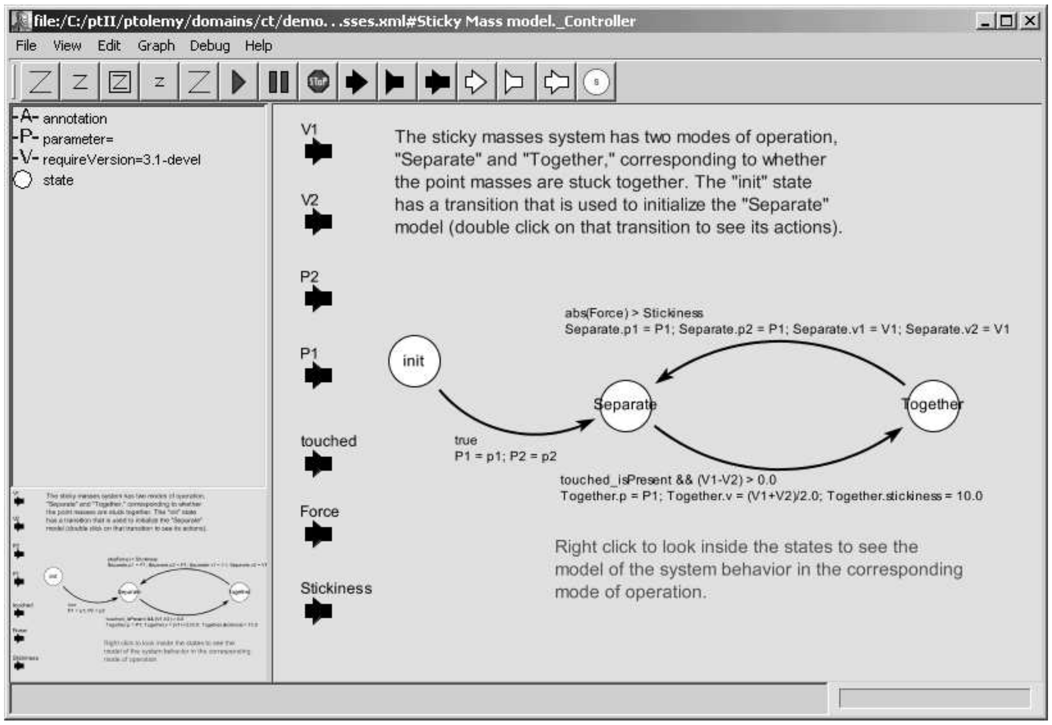
Authors: Edward A. Lee and Yuhong Xiong

FIGURE 5. Visual rendition of a Ptolemy II model as a block diagram in Vergil (in the DE domain).

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 5 and 6.

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 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.



The sticky masses system has two modes of operation, "Separate" and "Together," corresponding to whether the point masses are stuck together. The "init" state has a transition that is used to initialize the "Separate" model (double click on that transition to see its actions).

abs(Force) > Stickiness
Separate.p1 = P1; Separate.p2 = P1; Separate.v1 = V1; Separate.v2 = V1

true
P1 = p1; P2 = p2

touched_isPresent && (V1-V2) > 0.0
Together.p = P1; Together.v = (V1+V2)/2.0; Together.stickiness = 10.0

Right click to look inside the states to see the model of the system behavior in the corresponding mode of operation.

FIGURE 6. Visual rendition of a Ptolemy II model as a state transition diagram in Vergil (FSM domain).

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 7. 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.lib	This subpackage and its subpackages contain domain polymorphic actors. The actor.lib package is discussed further in section 6.3.
actor.parameters	This subpackage provides specialized parameters for specifying locations, ranges of values, etc.
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.
copernicus	This subpackage contains the “actor specialization” infrastructure (Java code generation).

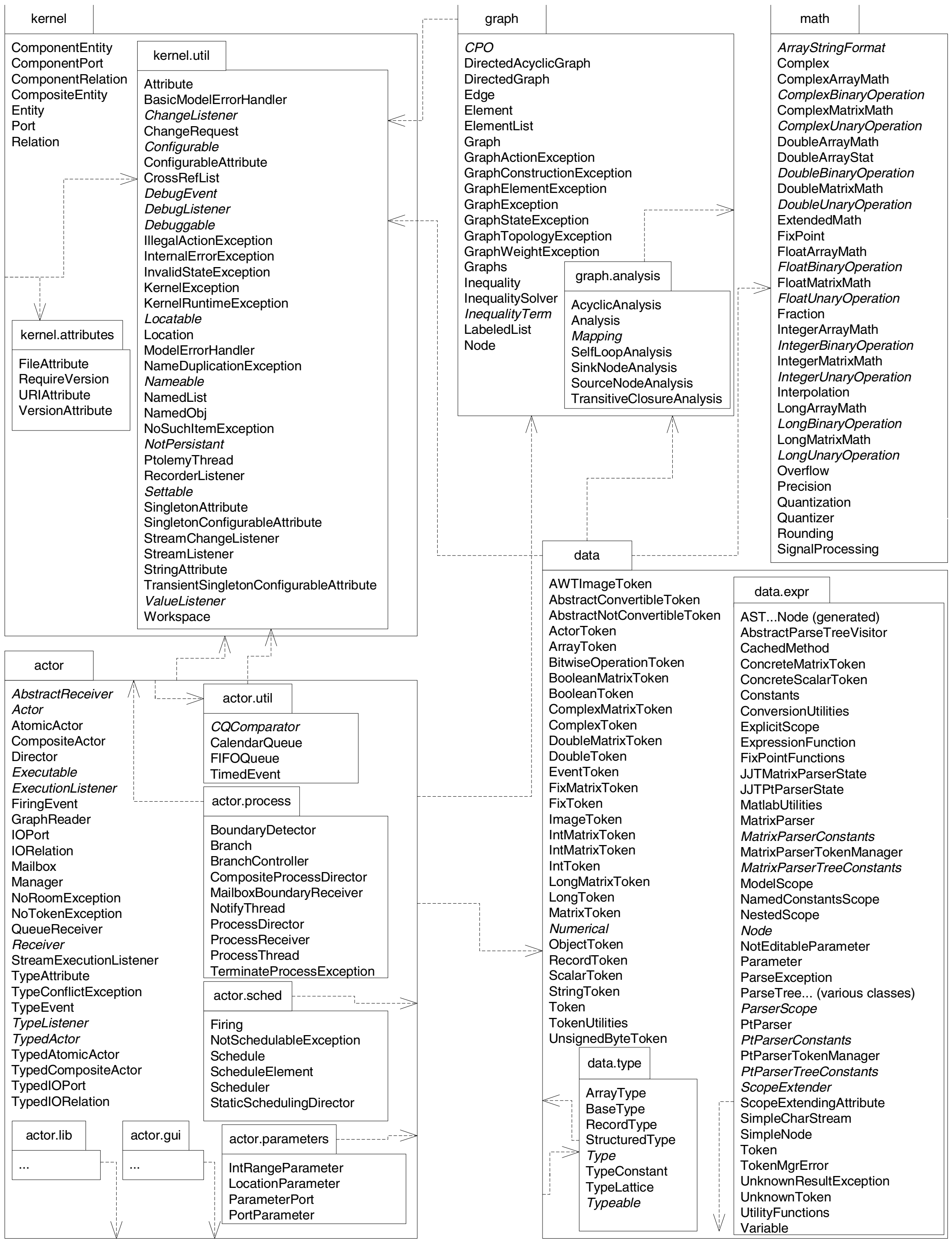


FIGURE 7. The core packages shown here support the data model (abstract syntax) and the actor model, (abstract semantics) of Ptolemy II designs.

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.
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.attributes	This subpackage of the kernel package provides specialized attributes such as FileAttribute, which is used in actors to specify a file or URL.
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.
matlab	This package contains the MATLAB interface.
util	This package contains various Ptolemy-independent utilities, such as string utilities and XML utilities.

6.1.1 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 8. 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 execution (starting, stopping, pausing). The director implements a semantics of a model of computation to govern the execution of actors contained by the CompositeActor.

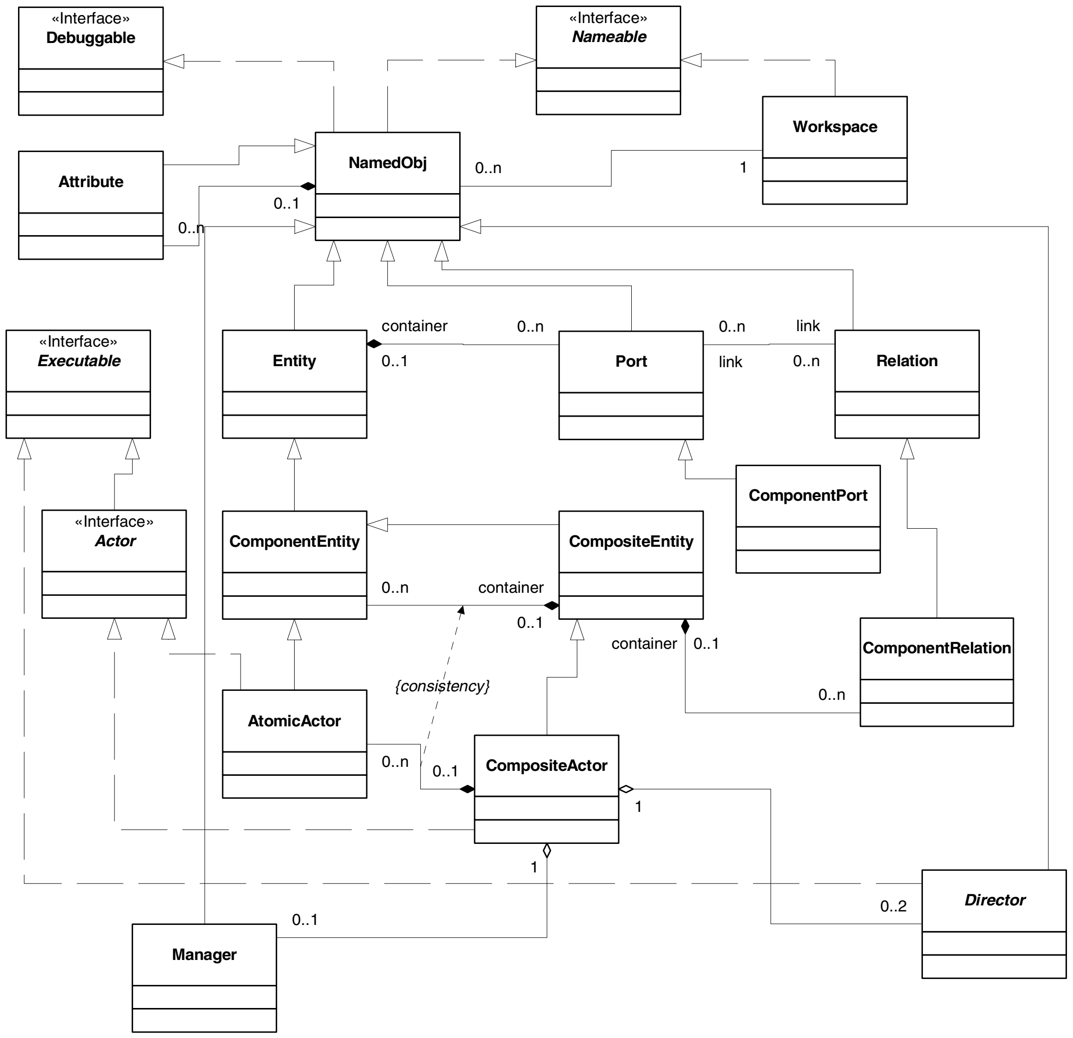


FIGURE 8. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and *actor* packages. They define the Ptolemy II abstract syntax and abstract semantics.

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.

6.2 Domains

The domains in Ptolemy II are subpackages of the ptolemy.domains package. The more mature and frequently used domains are shown in figure 9. The experimental domains and less commonly used domains are not shown, but the examples in figure 9 are illustrative of their structure. These pack-

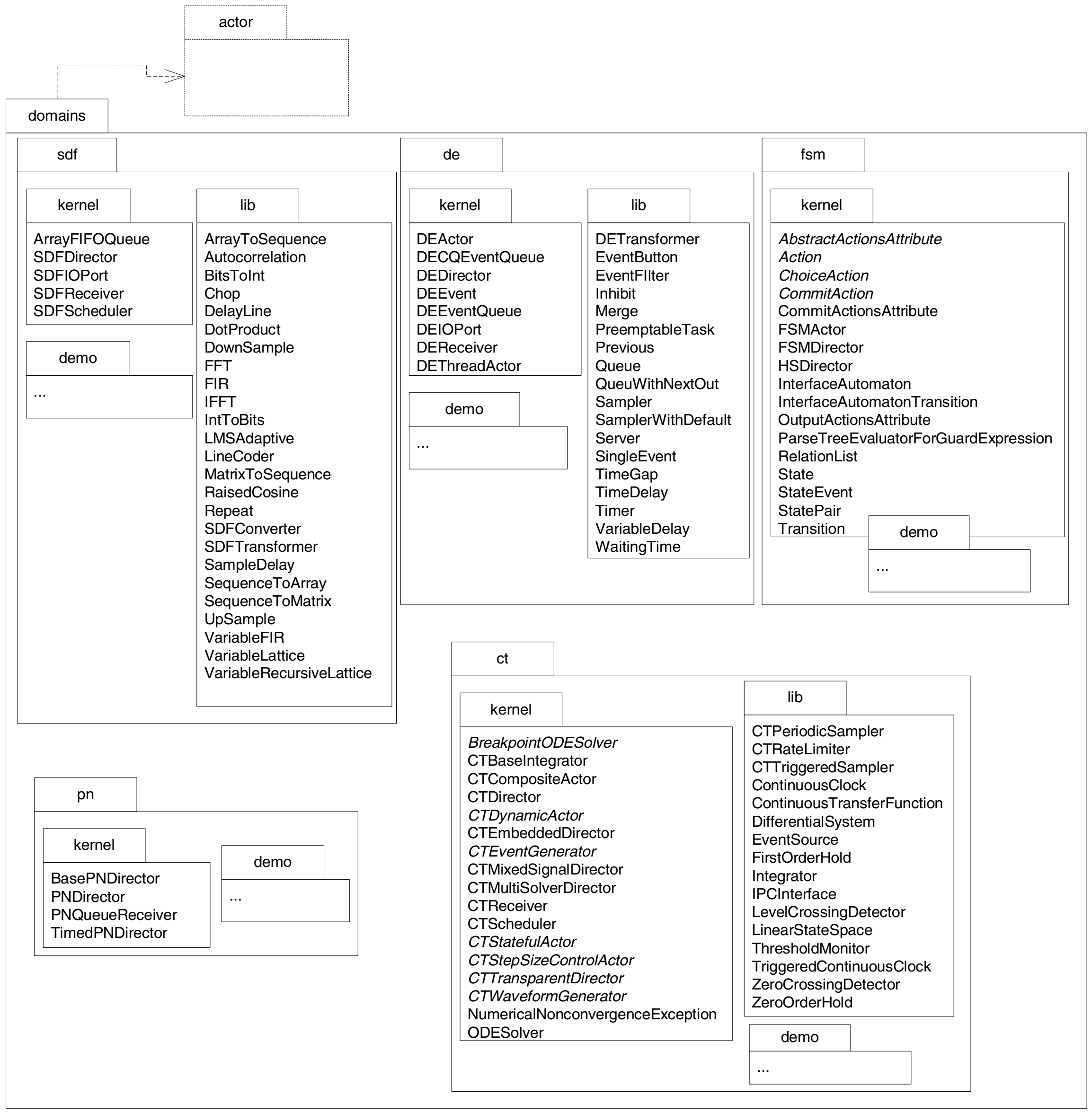


FIGURE 9. Package structure of common Ptolemy II domains.

ages 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.

6.3 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 10. These packages are briefly described below:

actor.lib	This package is the main library of polymorphic actors.
actor.lib.comm	This is a library of actors for modeling communications systems.
actor.lib.conversions	This package provides domain polymorphic actors that convert data between different types.
actor.lib.gui	This package is a library of polymorphic actors with user interface components, such as plotters.
actor.lib.hoc	This package is a library of higher-order components, which are components that construct portions of a model.
actor.lib.image	This package is a library of image processing actors.
actor.lib.io	This package provides file I/O.
actor.lib.io.comm	This package provides an actor that communicate via the serial ports. This actor works only under Windows.
actor.lib.jai	This is a library of image processing actors based on the *Java advanced imaging* API.
actor.lib.javasound	This package provides sound actors.
actor.lib.jmf	This is a library of image processing actors based on the *Java media framework* API.
actor.lib.joystick	This package provides an example actor that communicates with a particular I/O device, a joystick.
actor.lib.jxta	This is a library of experimental actors supporting the JXTA discovery mechanism from Sun Microsystems.
actor.lib.logic	This package provides actors that perform logical functions like AND, OR and NOT.
actor.lib.net	This package provides actors that communicate using datagrams.
actor.lib.python	This package provides an actor whose operation can be specified in Python.

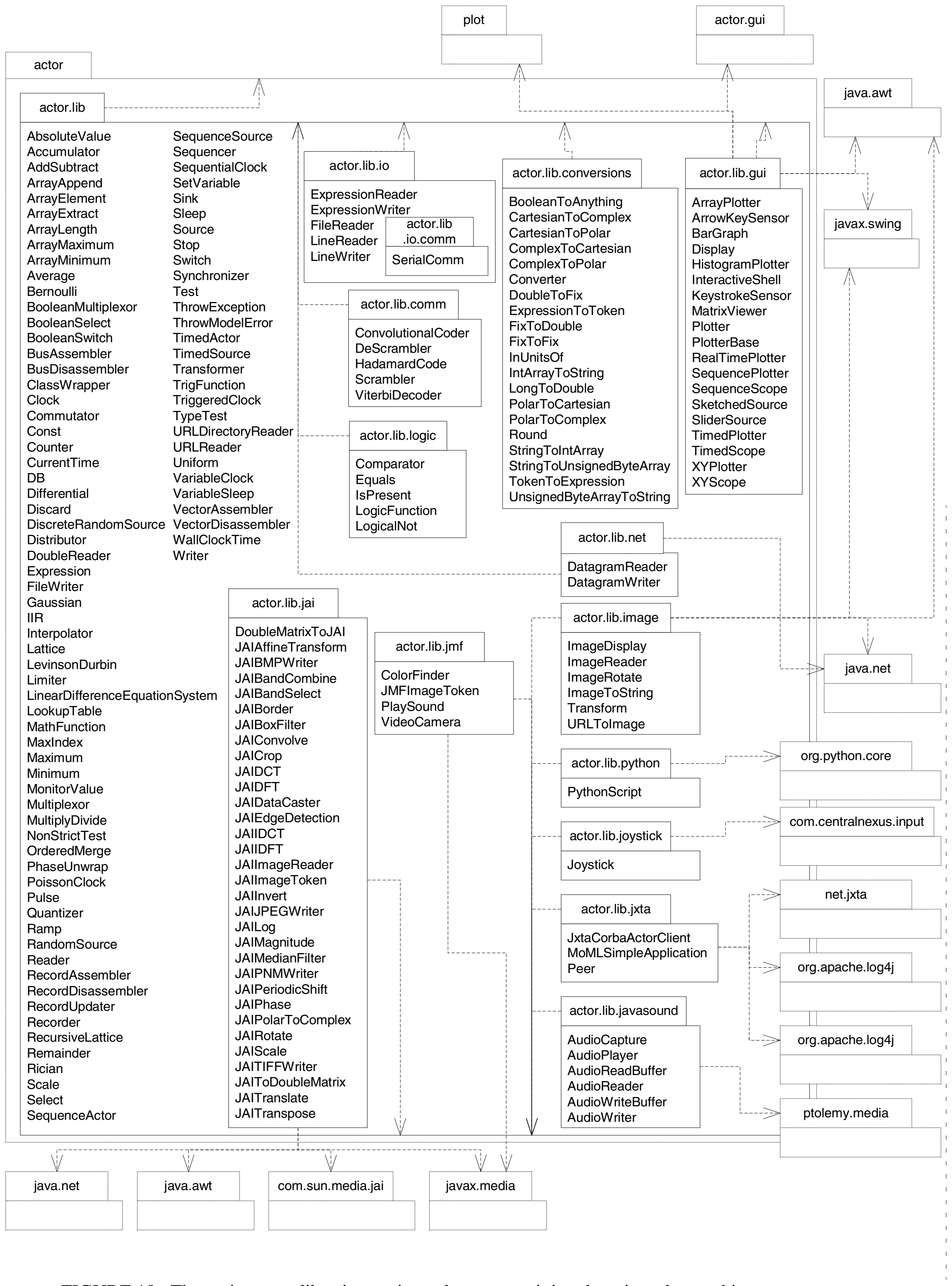


FIGURE 10. The major actor libraries are in packages containing domain-polymorphic actors.

6.4 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 figures 11 and 12. The intent of each package is described below:

actor.gui	This package 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.
moml.filter	This package provides backward compatibility between Ptolemy release. We hope to replace it with an XSL based solution in a future release.
plot	This package and its packages provides two-dimensional signal plotting widgets.
vergil	This package and its packages contains the Ptolemy II graphical user interface. It builds on Diva, a toolkit that extends Java 2D. For more information about Diva, see http://www.gigascale.org/diva

6.5 Capabilities

Ptolemy II is a third generation system. Its immediate predecessor, Ptolemy Classic, still has active users and developers, particularly through a commercial product that is based partly on it, Agilent's ADS. 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 [60]. 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, scalable, and understandable.
- *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.

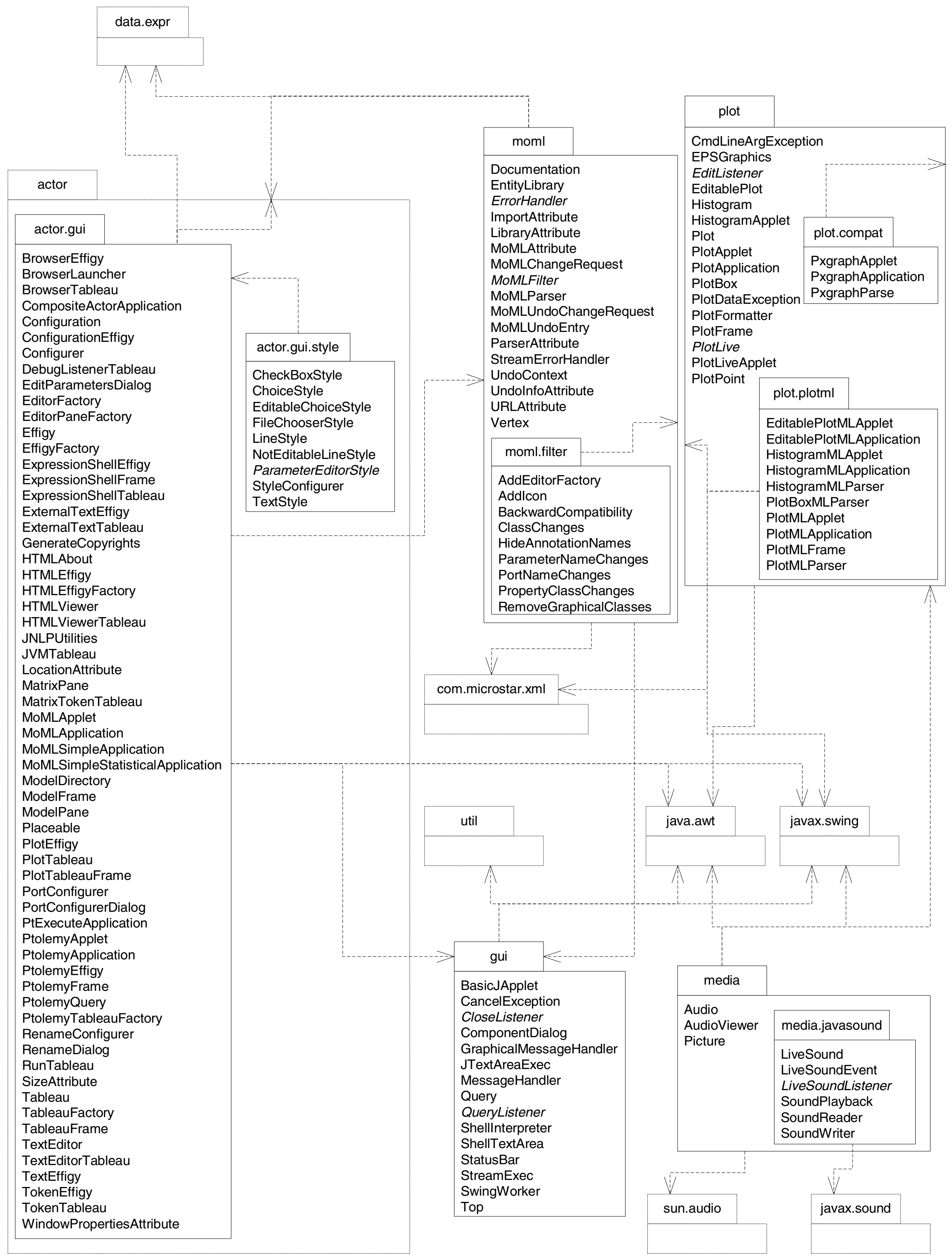


FIGURE 11. Packages in Ptolemy II that support the user interfaces, including the MoML XML schema, plotters and other display infrastructure, and support for windows and application configurations.

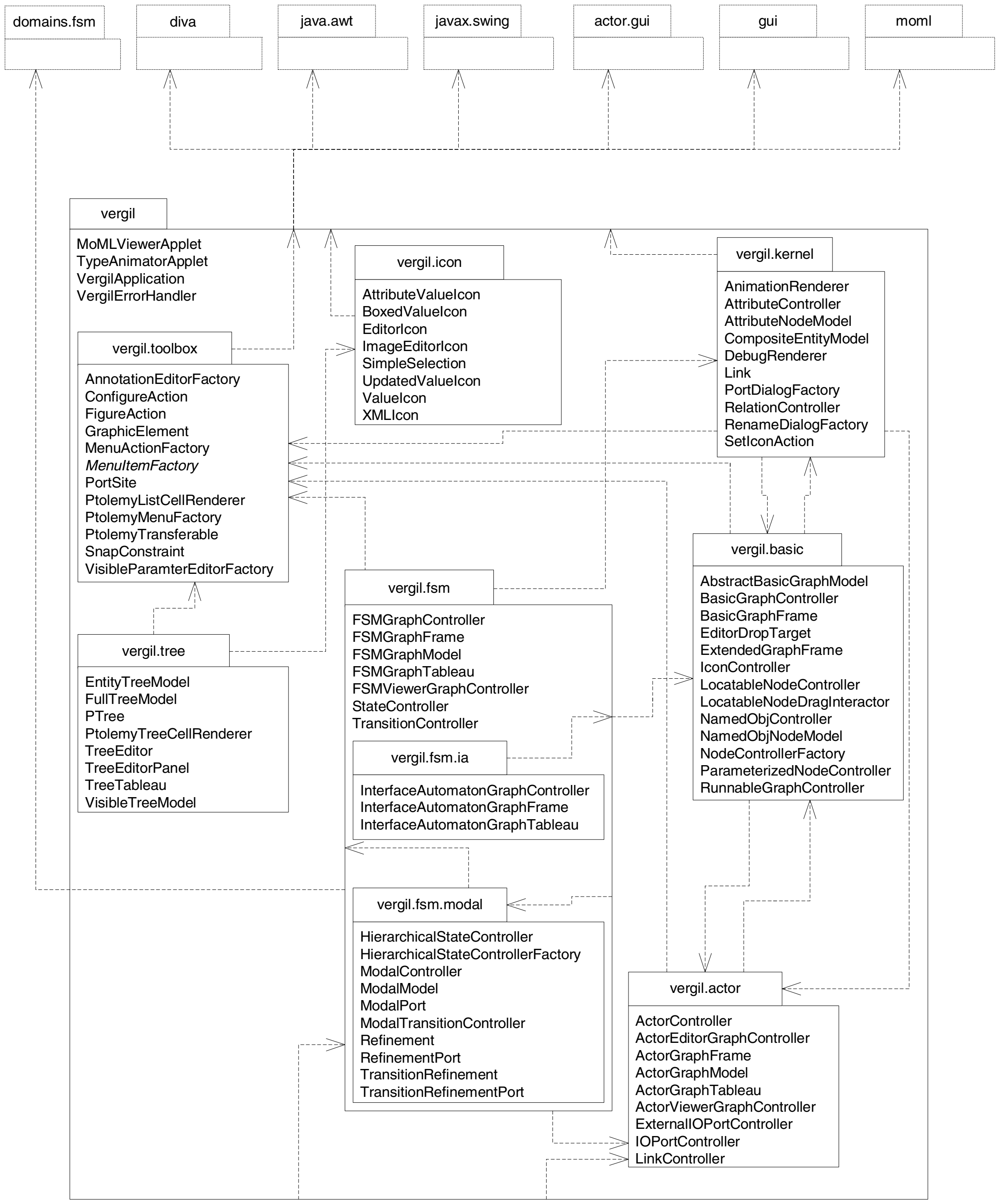


FIGURE 12. Packages in Ptolemy II provide the Vergil visual editor.

- *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 [51] 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 [89][110][113] and design pattern [35] 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 [107].
- *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 [92]. The type system is described in [124] and [125].
- *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*.
- *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.
- *Distributed models*. Ptolemy II has (still preliminary) infrastructure supporting distributed modeling using CORBA, Java RMI, or lower-level networking primitives. Ptolemy II has (still preliminary) support for migrating software components.
- *Component specialization.* 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 [98], [119] and [120].
- *Fully integrated expression language*. The Ptolemy II expression language is a higher-order, richly expressive language that is fully integrated with actor-oriented modeling. The type system inference mechanism propagates through expressions, parameters, and actor ports seamlessly.

6.6 Future Capabilities

Capabilities that we anticipate making available in the future include:

- *Integrated verification tools.* Modern verification tools based on model checking [46] 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.

7. Acknowledgements

There have been many participants in the Ptolemy Project over the years. So many, that we may (inadvertently) omit some names here. With apologies to those people, we list the contributors. The principal authors of version 3.0 of Ptolemy II and of this version of the overview document are

- Christopher Hylands,
- Edward Lee,
- Jie Liu,
- Xiaojun Liu,
- Stephen Neuendorffer,
- Yuhong Xiong,
- Yang Zhao, and
- Haiyang Zheng.

Major contributors to earlier versions include:

- John Davis, II,
- Chamberlain Fong,
- Mudit Goel,
- Bilung Lee,
- Lukito Muliadi,
- John Reekie,
- Neil Smyth, and
- Jeff Tsay.

Other contributors include Vincent Arnould, Frederic Boulanger, Adam Cataldo, Chris Chang, Albert Chen, Elaine Cheong, Brieuc Desoutter, Pedro Domecq, Johan Eker, Geroncio Galicia, Ben Horowitz, Heloise Hse, Jörn Janneck, Zoltan Kemenczy, Bart Kienhuis, Christoph Meyer Kirsch, Sanjeev Kohli, Robert Kroeger, David Lee, Michael Leung, John Li, Andrew Mihal, Eleftherios Matsikoudis, Aleksandar Necakov, Sarah Packman, Shankar Rao, Sonia Sachs, Michael Shilman, Sean Simmons, Mandeep Singh, Peter N. Steinmetz, Dick Stevens, Mary Stewart, Ned Stoffel, Manda Sutijono, Neil Turner, Guillaume Vibert, Kees Vissers, Brian K. Vogel, Yuke Wang, Xavier Warzee, Scott Weber, Paul Whitaker, Winthrop Williams, Ed Willink, Michael Wirthlin, William Wu, Paul Yang, James Yeh, Nick Zamora, and Rachel Zhou.

The Ptolemy Project has received financial support the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, Chess (the Center for Hybrid and Embedded Software Systems), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent, Atmel, Cadence, Hewlett-Packard, Hitachi, National Semiconductor, and Philips.

8. References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] G. Agha, “Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems,” in *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.

[3] G. Agha, “Concurrent object-oriented programming,” *Communications of the ACM*, 33(9):125–140, Sept. 1990.

[4] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, “Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[5] G. Agha, I. A. Mason, S. F.Smith, and C. L. Talcott, “A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] 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.

[7] G. R. Andrews, *Concurrent Programming — Principles and Practice*, Addison-Wesley, 1991.

[8] R. L. Bagrodia, “Parallel Languages for Discrete Event Simulation Models,” *IEEE Computational Science & Engineering*, vol. 5, no. 2, April-June 1998, pp 27-38.

[9] R. Bagrodia, R. Meyer, *et al.*, “Parsec: A Parallel Simulation Environment for Complex Systems,” *IEEE Computer*, vol. 31, no. 10, October 1998, pp 77-85.

[10] M. von der Beeck, “A Comparison of Statecharts Variants,” in *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, pp. 128-148, Springer-Verlag, 1994.

[11] 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.

[12] 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.

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

[14] S. Bhatt, R. M. Fujimoto, A. Ogielski, and K. Perumalla, “Parallel Simulation Techniques for Large-Scale Networks,” *IEEE Communications Magazine*, Vol. 36, No. 8, August 1998, pp. 42-47.

[15] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Mass, 1996.

[16] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E. A. Lee, “Gabriel: A Design Environment for DSP,” *IEEE Micro Magazine*, October 1990, vol. 10, no. 5, pp. 28-45.

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

[18] V. Bryant, “Metric Spaces,” Cambridge University Press, 1985.

[19] 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/publications/papers/94/JEurSim).

[20] A. Burns, *Programming in OCCAM 2*, Addison-Wesley, 1988.

[21] James C. Candy, “A Use of Limit Cycle Oscillations to Obtain Robust Analog-to-Digital Converters,” *IEEE Tr. on Communications*, Vol. COM-22, No. 3, pp. 298-305, March 1974.

[22] L. Cardelli, *Type Systems*, Handbook of Computer Science and Engineering, CRC Press, 1997.

[23] 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.

[24] K. M. Chandy and J. Misra, “Asynchronous Distributed Simulation Via a Sequence of Parallel Computations,” *Communications of the ACM*, vol. 24, no. 11, November 1981, pp. 198-205.

[25] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[26] 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.(http://ptolemy.eecs.berkeley.edu/publications/papers/00/concsys/)

[27] S. A. Edwards and E. A. Lee, “The Semantics and Execution of a Synchronous Block-Diagram Language,” **to appear** in *Science of Computer Programming*, 2003.

[28] 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/)

[29] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, "Taming Heterogeneity-the Ptolemy Approach," *Proceedings of the IEEE*, V. 91, No 1, January 2003.

[30] P. H. J. van Eijk, C. A. Vissers, M. Diaz, *The formal description technique LOTOS*, Elsevier Science, B.V., 1989. (http://wwwtios.cs.utwente.nl/lotos)

[31] P. A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, 1995.

[32] 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.(http://ptolemy.eecs.berkeley.edu/publications/papers/00/dt/)

[33] M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 1997.

[34] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, October 1990, pp 30-53.

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

[36] C. W. Gear, "Numerical Initial Value Problems in Ordinary Differential Equations," Prentice Hall Inc. 1971.

[37] A. J. C. van Gemund, *"Performance Prediction of Parallel Processing Systems: The PAMELA Methodology,"* Proc. 7th Int. Conf. on Supercomputing, pages 418-327, Tokyo, July 1993.

[38] 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/publications/papers/98/starcharts)

[39] 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. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/PNinPtolemyII)

[40] M. Grand, *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley & Sons, 1998.

[41] C. Hansen, "Hardware logic simulation by compilation," In *Proceedings of the Design Automation Conference* (DAC). SIGDA, ACM, 1988.

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

[43] P. G. Harrison, "A Higher-Order Approach to Parallel Algorithms," *The Computer Journal*, Vol. 35, No. 6, 1992.

[44] T. A. Henzinger, B. Horowitz and C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming," EMSOFT 2001, Tahoe City, CA, Springer-Verlag,

[45] 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.

[46] 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.

[47] T. A. Henzinger and C. M. Kirsch, "The Embedded Machine: Predictable, portable real-time code," In *Proceedings of Conference on Programming Language Design and Implementation* (PLDI). SIGPLAN, ACM, June 2002.

[48] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, 8(3):323–363, June 1977.

[49] M. G. Hinchey and S. A. Jarvis, *Concurrent Systems: Formal Developments in CSP*, McGraw-Hill, 1995.

[50] C. W. Ho, A. E. Ruehli, and P. A. Brennan, "The Modified Nodal Approach to Network Analysis," IEEE Tran. on Circuits and Systems, Vol. CAS-22, No. 6, 1975, pp. 504-509.

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

[52] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice-Hall, 1985.

[53] IEEE DASC 1076.1 Working Group, "VHDL-A Design Objective Document, version 2.3," http://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt

[54] D. Jefferson, Brian Beckman, et al, "Distributed Simulation and the Time Warp Operating System," UCLA Computer Science Department: 870042, 1987.

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

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

[57] E. Kohler, *The Click Modular Router*, Ph.D. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2001.

[58] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.

[59] P. Laramie, R.S. Stevens, and M.Wan, "Kahn process networks in Java," ee290n class project report, Univ. of California at Berkeley, 1996.

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

[61] B. Lee and E. A. Lee, "Interaction of Finite State Machines with Concurrency Models," *Proc. of Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/Interaction-FSM/)

[62] B. Lee and E. A. Lee, "Hierarchical Concurrent Finite State Machines in Ptolemy," *Proc. of International Conference on Application of Concurrency to System Design*, p. 34-40, Fukushima, Japan, March 1998 (http://ptolemy.eecs.berkeley.edu/publications/papers/98/HCFSMinPtolemy/)

[63] E. A. Lee, S. Neuendorffer and M. J. Wirthlin, “Actor-Oriented Design of Embedded Hardware and Software Systems,” **invited paper, to appear in** *Journal of Circuits, Systems, and Computers*, 2003.

[64] E. A. Lee, “Embedded Software,” in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[65] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” in *Readings in Hardware/Software Co-Design*, G. De Micheli, R. Ernst, and W. Wolf, eds., Morgan Kaufmann, San Francisco, 2002 (reprinted from 70).

[66] E. A. Lee, “What's Ahead for Embedded Software?” *IEEE Computer*, September 2000, pp. 18-26.

[67] E. A. Lee, “Modeling Concurrent Real-time Processes Using Discrete Events,” Invited paper to *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, Volume 7, 1999, pp 25-45. Also UCB/ERL Memorandum M98/7, March 4th 1998.(http://ptolemy.eecs.berkeley.edu/publications/papers/98/realtime)

[68] E. A. Lee and Y. Xiong, “System-Level Types for Component-Based Design,” *First Workshop on Embedded Software*, EMSOFT 2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001. (also Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, CA 94720, USA, February 29, 2000. http://ptolemy.eecs.berkeley.edu/publications/papers/01/systemLevelType/).

[69] E. A. Lee, “Computing for Embedded Systems,” **invited paper**, *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.

[70] 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/publications/papers/95/processNets)

[71] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,”, *IEEE Transactions on CAD*, Vol 17, No. 12, December 1998 (Revised from ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997). (http://ptolemy.eecs.berkeley.edu/publications/papers/97/denotational/)

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

[73] M. A. Lemkin, *Micro Accelerometer Design with Digital Feedback Control*, Ph.D. dissertation, University of California, Berkeley, Fall 1997.

[74] S. Y. Liao, S. Tjiang, and R. Gupta, “An efficient implementation of reactivity for modeling hardware in the Scenic design environment,” In *Proceedings of the 34th Design Automation Conference* (DAC’1997). SIGDA, ACM, 1997.

[75] J. Liu, J. Eker, J. W. Janneck and E. A. Lee, “Realistic Simulations of Embedded Control Systems,” *International Federation of Automatic Control*, *15th IFAC World Congress*, Barcelona, Spain, July 21-26, 2002.

[76] J. Liu, X. Liu, and E. A. Lee, “Modeling Distributed Hybrid Systems in Ptolemy II,” **invited embedded tutorial** in *American Control Conference*, Arlington, VA, June 25-27, 2001.

[77] J. Liu, S. Jefferson, and E. A. Lee, “Motivating Hierarchical Run-Time Models in Measurement and Control Systems,” *American Control Conference*, Arlington, VA, pp. 3457-3462, June 25-27, 2001.

[78] J. Liu and E. A. Lee, “A Component-Based Approach to Modeling and Simulating Mixed-Signal and Hybrid Systems,” **to appear** in *ACM Trans. on Modeling and Computer Simulation,* special issue on computer automated multi-paradigm modeling, 2003.

[79] J. Liu and E. A. Lee, “On the Causality of Mixed-Signal and Hybrid Models,” *6th International Workshop on Hybrid Systems: Computation and Control* (HSCC '03), April 3-5, Prague, Czech Republic, 2003.

[80] J. Liu, "Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems," **Ph.D. thesis**, Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, CA 94720, December 20th, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/responsibleFrameworks/)

[81] 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. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/MixedSignalinPtII/)

[82] J. Liu and E. 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

[83] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee, "A Hierarchical Hybrid System and Its Simulation", 1999 38th IEEE Conference on Decision and Control (CDC'99), Phoenix, Arizona.

[84] X. Liu, J. Liu, J. Eker, and E. A. Lee, “Heterogeneous Modeling and Design of Control Systems,” **to appear** in *Software-Enabled Control: Information Technology for Dynamical Systems*, T. Samad and G. Balas (eds.), New York City: IEEE Press, 2003.

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

[86] 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.

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

[88] K. Mehlhorn and Stefan Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1997.

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

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

[91] R. Milner, *“A Calculus of Communicating Systems”*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

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

[93] J. Misra, “Distributed Discrete-Event Simulation,” *Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39-65.

[94] L. Muliadi, "Discrete Event Modeling in Ptolemy II," MS Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/deModeling/)

[95] P. K. Murthy and E. A. Lee, “Multidimensional Synchronous Dataflow,” *IEEE Transactions on Signal Processing*, volume 50, no. 8, pp. 2064 -2079, August 2002.

[96] L. W. Nagal, “SPICE2: A Computer Program to Simulate Semiconductor Circuits,” ERL Memo No. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA 94720.

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

[98] S. Neuendorffer, “Automatic Specialization of Actor-Oriented Models in Ptolemy II,” Master's Report, Technical Memorandum UCB/ERL M02/41, University of California, Berkeley, CA 94720, December 25, 2002.(http://ptolemy.eecs.berkeley.edu/papers/02/actorSpecialization)

[99] A. R. Newton and A. L. Sangiovanni-Vincentelli, “Relaxation-Based Electrical Simulation,” *IEEE Tr. on Electronic Devices*, Vol. ed-30, No. 9, Sept. 1983.

[100]S. Oaks and H. Wong, *Java Threads,* O’Reilly, 1997.

[101]OMG, *Unified Modeling Language: Superstructure*, version 2.0, 3rd revised submission to RFP ad/00-09-02, April 10, 2003

[102]J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

[103]J. K. Ousterhout, *Scripting: Higher Level Programming for the 21 Century*, IEEE Computer magazine, March 1998.

[104]T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. **Ph.D. Dissertation**. EECS Department, University of California. Berkeley, CA 94720, December 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/)

[105]J. K. Peacock, J. W. Wong and E. G. Manning, “Distributed Simulation Using a Network of Processors,” *Computer Networks*, vol. 3, no. 1, February 1979, pp. 44-56.

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

[107]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.(http://ptolemy.eecs.berkeley.edu/publications/papers/99/sftwareprac/)

[108]J. Rehof and T. Mogensen, “Tractable Constraints in Finite Semilattices,” *Third International Static Analysis Symposium,* pp. 285-301, Volume 1145 of Lecture Notes in Computer Science, Springer, Sept., 1996.

[109]C. Rettig, “Automatic Units Tracking,” *Embedded System Programming*, March, 2001.

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

[111]R. C. Rosenberg and D.C. Karnopp, *Introduction to Physical System Dynamics*, McGraw-Hill, NY, 1983.

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

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

[114]J. Rumbaugh, *OMT Insights*, SIGS Books, 1996.

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

[116]B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY 1994.

[117]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. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/CSPinPtolemyII/)

[118]J. Teich, E. Zitzler, and S. Bhattacharyya, “3D exploration of software schedules for DSP algorithms,” In *Proceedings of International Symposium on Hardware/Software Codesign* (CODES). SIGDA, ACM, May 1999.

[119]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).

[120]J. Tsay, C. Hylands and E. A. Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[121]P. Whitaker, "The Simulation of Synchronous Reactive Systems In Ptolemy II," Master's Report, Memorandum UCB/ERL M01/20, Electronics Research Laboratory, University of California, Berkeley, May 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/sr/)

[122]World Wide Web Consortium, *XML 1.0 Recommendation*, October 2000, http://www.w3.org/XML/

[123]World Wide Web Consortium, *Overview of SGML Resources*, August 2000, http://www.w3.org/MarkUp/SGML/

[124]Y. Xiong and E. 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.

[125]Y. Xiong, "An Extensible Type System for Component-Based Design," **Ph.D. thesis**, Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002. (http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem).