DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

UNIVERSITY OF CALIFORNIA

BERKELEY, CALIFORNIA 94720

ANNUAL REPORT

HETEROGENEOUS MODELING AND DESIGN

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY (DARPA)

COMPOSITE CAD PROGRAM

CONTRACTOR:	**University of California at Berkeley**
AGREEMENT NUMBER:	**DAAB07-97-C-J007**
CONTRACT PERIOD:	**11/18/96 - 11/31/99**
DATE:	**May 31, 1999**
TITLE:	**Heterogeneous Modeling And Design**
REPORT PERIOD:	**11/15/97 - 11/15/98**
SPONSOR:	**Air Force Research Laboratory (AFRL)**
TECHNICAL POC:	**James P. Hanna**
REPORT PREPARED BY:	**Edward A. Lee**

Principal Investigator: Edward A. Lee
Organization: University of California at Berkeley

Contents

1. Project Overview

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

1.1 MODELING AND DESIGN

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 ARCHITECTURE DESIGN

Architecture description languages (ADLs), such as Wright [27] and Rapide [45], 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 [27], 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 [27], we use a technique much more powerful than type checking alone, namely polymorphism.

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

Third, whereas an ADL might choose a particular model of computation to provide it with a formal structure, such as CSP for Wright [27], we have developed a more abstract formal framework that describes models of computation at a meta level [2]. 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 [27].

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

1.3 MODELS OF COMPUTATION

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

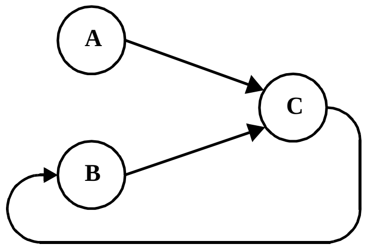

FIGURE 1. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

1.3.1 Communicating Sequential Processes - CSP

In the CSP domain (communicating sequential processes), created by Neil Smyth [12], 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) [39] and Milner’s *calculus of communicating systems* (CCS) [49]. This model of computation has been realized in a number of concurrent programming languages, including Lotos and Occam.

Rendezvous models are particularly well-matched to applications where resource sharing is a key element, such as client-server database models and multitasking or multiplexing of hardware resources. A key feature of rendezvous-based models is their ability to cleanly model nondeterminate interactions. The CSP domain implements both conditional send and conditional receive. It also includes an experimental timed extension.

1.3.2 Continuous Time - CT

In the CT domain (continuous time), created Jie Liu [13], 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.

Since many solvers iterate to a fixed point solution, the CT domain has stressed the Ptolemy II infrastructure to ensure that it supports speculative computation and rollback.

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

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

1.3.3 Discrete-Events - DE

In the discrete-event (DE) domain, created by Lukito Muliadi, the actors communicate via sequences of events placed in time, along a real time line. An *event* consists of a *value* and *time stamp*.

Actors can either be processes that react to events (implemented as Java threads) or functions that fire when new events are supplied. This model of computation is popular for specifying digital hardware and for simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog.

DE models are excellent descriptions of concurrent hardware, although increasingly the globally consistent notion of time is problematic. In particular, it over-specifies (or over-models) systems where maintaining such a globally consistent notion is difficult, including large VLSI chips with high clock rates. Every event is placed precisely on a globally consistent time line.

The DE domain implements a fairly sophisticated discrete-event simulator. DE simulators in general need to maintain a global queue of pending events sorted by time stamp (this is called a *priority queue*). This can be fairly expensive, since inserting new events into the list requires searching for the right position at which to insert it. The DE domain uses a calendar queue data structure [31] 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.

1.3.4 Distributed Discrete Events - DDE

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

1.3.5 Discrete Time - DT

The discrete-time (DT) domain, which has not been written yet, will extend 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, will be supported.

1.3.6 Finite-State Machines - FSM

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

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

FSM models have a number of key weaknesses. First, at a very fundamental level, they are not as expressive as the other models of computation described here. They are not sufficiently rich to describe all 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.

The latter problem can often be solved by using FSMs in combination with concurrent models of computation. This was first noted by David Harel, who introduced that Statecharts formalism. Statecharts combine a loose version of synchronous-reactive modeling (described below) with FSMs [36]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [37].

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 [11]. 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 [56].

1.3.7 Process Networks - PN

In the process networks (PN) domain, created by Mudit Goel [14], 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 [40].

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 [41], 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.

1.3.8 Synchronous Dataflow - SDF

The synchronous dataflow (SDF) domain, created by Steve Neuendorffer, handles regular computations that operate on streams. Dataflow models, popular in signal processing, are a special case of process networks (for the complete explanation of this, see [43]). 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.

1.3.9 Synchronous/Reactive - SR

In the synchronous/reactive (SR) model of computation [28], 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 [30], Signal [29], Lustre [32], and Argos [46].

SR models are excellent for applications with concurrent and complex control logic. Because of the tight synchronization, safety-critical real-time applications are a good match. However, also because of the tight synchronization, some applications are overspecified in the SR model, limiting the implementation alternatives. Moreover, in most realizations, modularity is compromised by the need to seek a global fixed point at each clock tick. An SR domain has not yet been implemented in Ptolemy II, although the methods used by Stephen Edwards in Ptolemy Classic can be adapted to this purpose [33].

1.4 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 [2].

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

Another alternative would be to choose one concurrent model of computation, say the rendezvous model, and show that all the others are subsumed as special cases. This is relatively easy to do, in theory. It is the premise of Wright, for example [27]. Most of these models of computation are sufficiently expressive to be able to subsume most of the others. However, this fails to acknowledge the strengths and weaknesses of each model of computation. Rendezvous is very good at resource management, but very awkward for loosely coupled data-oriented computations. Asynchronous message passing is the reverse, where resource management is awkward, but data-oriented computations are natural[1]. Thus, to design interesting systems, designers need to use heterogeneous models.

1.5 VISUAL SYNTAXES

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

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

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

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

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

1.6 PTOLEMY II

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.

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.

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.

1.6.1 Package Structure

The package structure is shown in figure 2. 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.event* which depends on *kernel* which depends on *kernel.util*. *Actor* also depends on *data* and *graph*. The role of each package is explained below.

actor	This package supports executable entities that receive and send data through ports. It includes both untyped and typed actors. For typed actors, it implements a sophisticated type system that supports polymorphism. It includes the base class Director for domain-specific classes that control the execution of a model.
actor.gui	This subpackage is a library of polymorphic actors with user interface components, plus some convenience base classes for applets and applications.
actor.lib	This subpackage is a library of polymorphic actors.
actor.process	This subpackage provides infrastructure for domains where actors are processes implemented on top of Java threads.
actor.sched	This subpackage provides infrastructure for domains where actors are statically scheduled by the director.
actor.util	This subpackage contains utilities that support directors in various domains. Specifically, it contains a simple FIFO Queue and a sophisticated priority queue called a calendar queue.
data	This package provides classes that encapsulate and manipulate data that is transported between actors in Ptolemy models.
data.expr	This class supports an extensible expression language and an interpreter for that language. Parameters can have values specified by expressions. These expressions 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.
domains	This package contains one subpackage for each Ptolemy II domain.
graph	This package provides algorithms for manipulating and analyzing mathematical graphs. Mathematical graphs are simpler than Ptolemy II clustered graphs in that there is no hierarchy, and arcs link exactly two nodes. This package is expected to supply a growing library of algorithms.
gui	This package contains generically useful user interface components.
kernel	This package provides the software architecture for the key abstract syntax, clustered graphs. The classes in this package support entities with ports, and relations that connect the ports. Clustering is where a collection of entities is encapsulated in a single composite entity, and a subset of the ports of the inside entities are exposed as ports of the cluster entity.
kernel.event	This package contains classes and interfaces that support controlled mutations of clustered graphs. Mutations are modifications in the topology, and in general, they are permitted to occur during the execution of a model. But in certain domains,

where maintaining determinacy is imperative, the director may wish to exercise tight control over precisely when mutations are performed. This package supports queueing of mutation requests for later execution. It uses a publish-and-subscribe

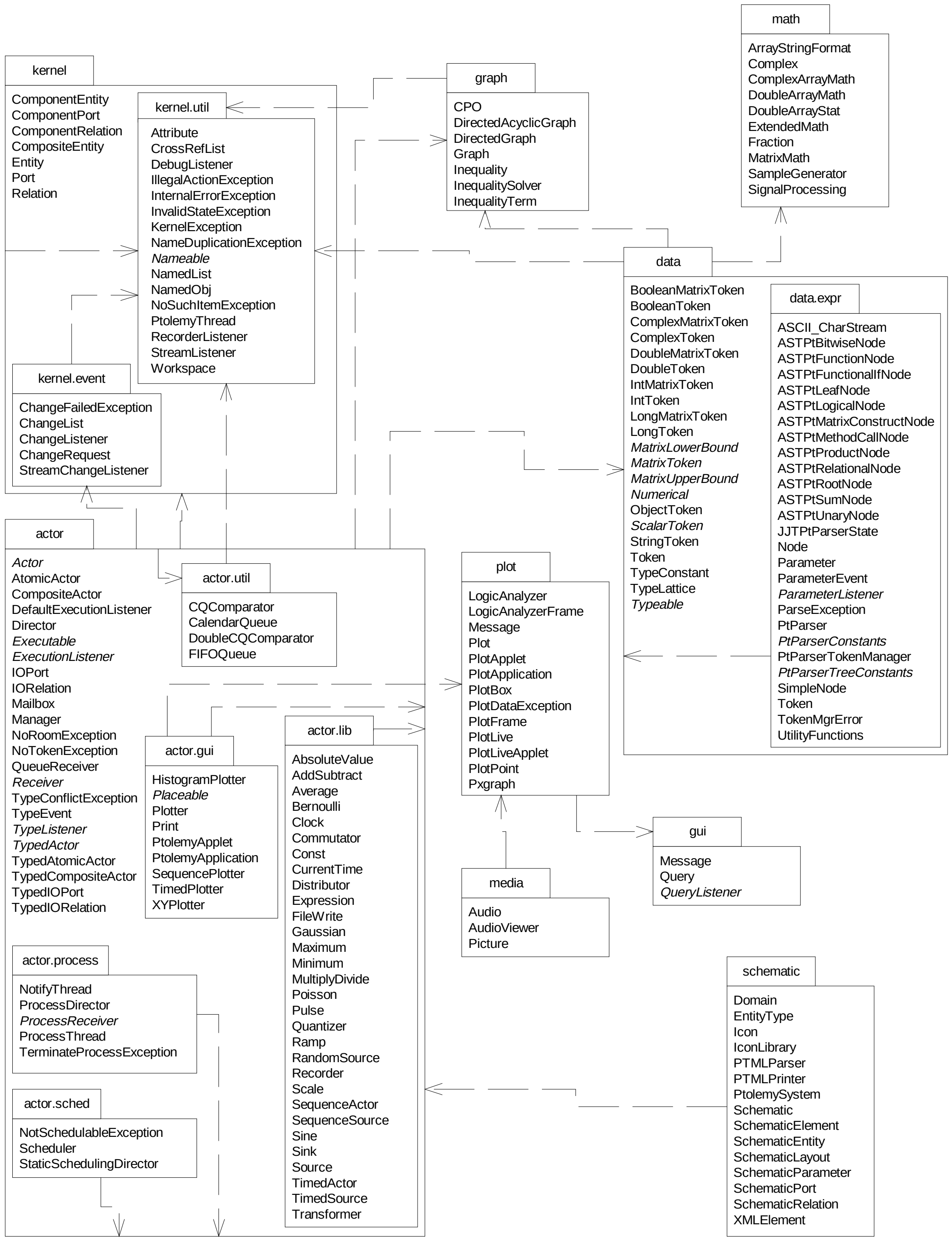

FIGURE 2. The package structure of Ptolemy II, without the domains.

	design pattern.
kernel.util	This subpackage of the kernel package provides a collection of utility classes that do not depend on the kernel package. It is separated into a subpackage so that these utility classes can be used without the kernel. The utilities include a collection of exceptions, classes supporting named objects with attributes, lists of named objects, a specialized cross-reference list class, and a thread class that helps Ptolemy keep track of executing threads.
math	This package encapsulates mathematical functions and methods for operating on matrices and vectors. It also includes a complex number class and a class supporting fractions.
media	This package encapsulates a set of classes supporting audio and image processing.
plot	This package provides two-dimensional signal plotting widgets.
schematic	This package provides a top-level interface to Ptolemy II. A GUI can use the classes in this package to gain access to Ptolemy II models.

1.6.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 3. This is a *static structure diagram* in UML (unified modeling language). The key syntactic elements are boxes, which represent classes, the hollow arrow, which indicates generalization, and other lines, which indicate association. Some lines have a small diamond, which indicates aggregation. The syntax of this diagram and 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 will be fully explained in the kernel chapter. ComponentPort, ComponentRelation, and ComponentEntity extend these classes by adding support for clustered graphs. CompositeEntity extends ComponentEntity and represents an aggregation of instances of ComponentEntity and ComponentRelation.

The Executable interface, 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.

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

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

1.6.3 Capabilities

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

- *Higher level concurrent design in Java*^{TM}. Java support for concurrent design is very low level, based on threads and monitors. Maintaining safety and liveness can be quite difficult [42]. 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.
- *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 mecha-

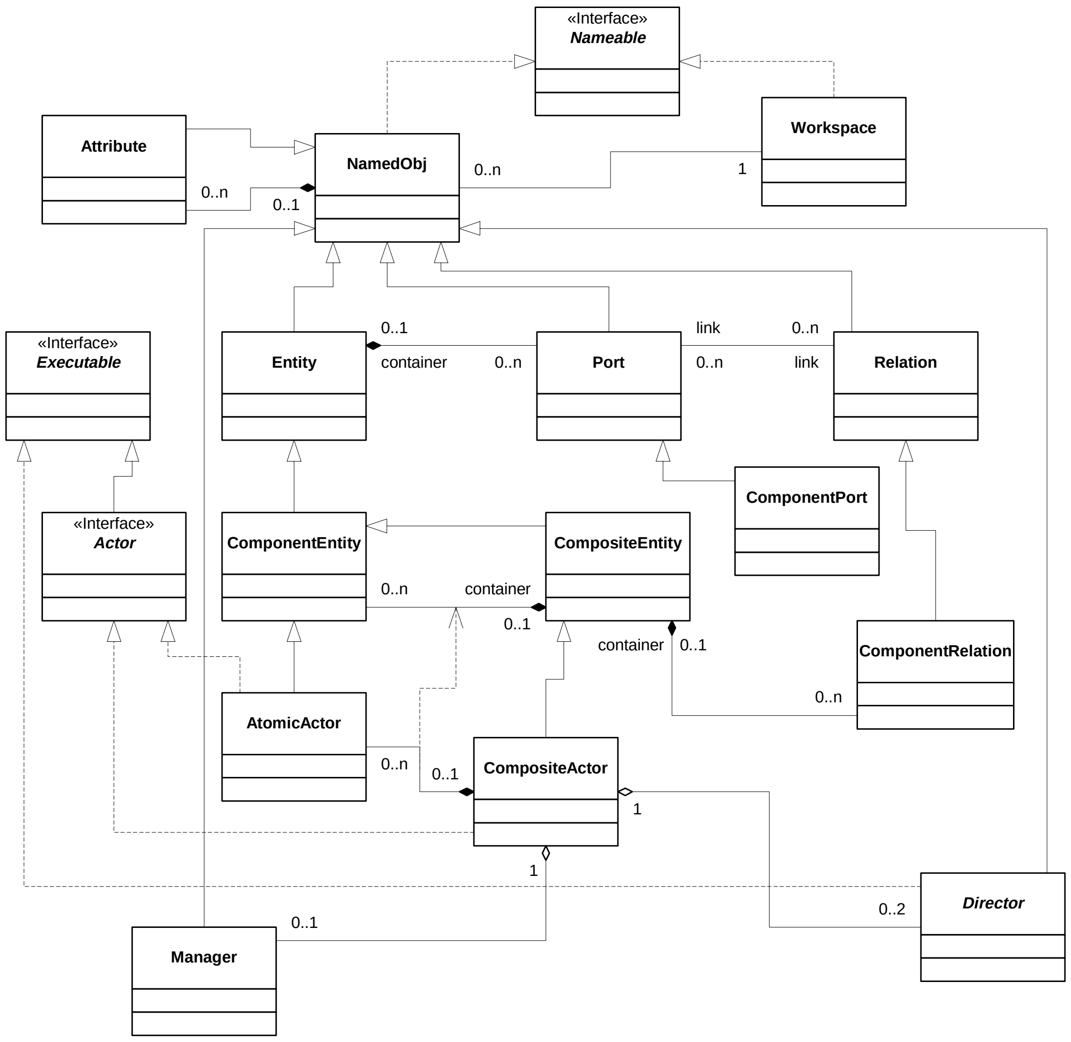

FIGURE 3. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and

nisms that attach semantics (such as dataflow, analog circuits, finite-state machines, etc.) to the graphs.

- *Improved heterogeneity.* 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 [39] 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 [48][53][55] and design pattern [34] 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 [52].
- *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 [49].
- *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*.

2. Summary of Accomplishments

The major accomplishments of the project are summarized in this section. Concrete deliverables included monthly and annual reports, software, demonstrable technology transfer, and numerous publications, all of which have been posted on the web.

2.1 TASK 1: MODULAR DEPLOYABLE DESIGN TOOLS

2.1.1 Synchronization Infrastructure

Ptolemy II is multithreaded at its core. Its design assumes that multiple threads will be interacting simultaneously within a Ptolemy model, or externally with other tools or interfaces. We have constructed a sophisticated synchronization mechanism that we call the *workspace* which makes it safe for multiple threads to interact with the same data structures. This mechanism permits any number of threads to simultaneously hold read permission on a workspace, and at most one thread to hold write permission on the workspace. No thread can hold write permission if any other thread holds read or write permission. Priority is given to threads wanting write permission, so that when data is read, it is as up-to-date as possible.

2.1.2 Clustered Graphs

The kernel package in Ptolemy II supports an abstract syntax called "clustered graphs" for constructing modular models. In particular, it introduces the notion of *transparent* and *opaque* ports, which serve to hierarchically expose subsystem abstractions in a controlled way. Inferences such as the width of connections (the number of channels) are automatically carried across these hierarchical boundaries, as are type constraints.

2.1.3 Type System

The sophisticated Ptolemy II type system, built by Yuhong Xiong, is loosely based on concepts first embodied in the ML language. His design permits subsystems (including atomic actors) to define constraints on type relationships between their inputs and outputs. An algorithm based on partial orders resolves types so that all constraints are satisfied, or the constraints are determined to not be satisfiable. His design supports a rich variety of polymorphic actors, and meshes with the actor package to support mutable systems.

Yuhong's design adds a method typeConstraints() to the Actor interface. The default implementation in the AtomicActor class supports a broad class of polymorphic actors where the input type are required to be at least as specific as the output types. The default implementation for the CompositeActor class collects type constraints from contained actors and forms constraints for the connections.

TypedIOPort is a new subclass of IOPort that stores the declared and resolved types for ports. The Director class has a method resolveType() that solves the type constraints by calling a generic inequality solver.

2.1.4 Actor Package

The actor package, which provides basic support for flow of control and message passing, continues to mature. It provides base classes for actors, their ports, the directors (which manage the execution of a domain-specific model), and a manager (which manages the overall execution in a domain-independent way). A significant challenge was to ensure that multi-threaded models are controlled systematically from a central controller, and that exceptions are handled cleanly.

The actor package includes an API for handling of time in a unified way across domains, including those that do not have explicit notions of time in their semantics. This model permits a director in a domain to suspend execution while requesting re-invocation at a specified time in the future. Even if the parent domain does not include time in its semantics, as long as some domain above it in the hierarchy does, then the director will be re-invoked as requested. The notion of time is kept consistent across all timed domains without requiring untimed domains to explicitly manage it.

2.1.5 Data Encapsulation

The Ptolemy II data package contains a set of classes that wrap data exchanged between actors. A *token* is a unit of data in this package. It is *immutable*, in that once created, its data value can never be changed. This design proves especially convenient in a multithreaded environment, where it obviates the need to synchronize threads that are modifying and/or reading data values. However, Jeff Tsay showed that it comes at a non-trivial cost in performance.

2.1.6 Expression Language

Neil Smyth developed an expression parser for the data package that uses JavaCC (informally called Jack), a "compiler-compiler" from Sun Microsystems. The expression parser is designed to support interrelated parameters (via a publish-and-subscribe model) and dynamically evaluated expressions in actors. This represents a major improvement over the expression parser in Ptolemy 0.

The intended use of the expression language is for specifying interlinked parameter values in a model, for specifying guards on transitions in state machines, and for providing a convenient, interpreted scripting language for quick construction of actors.

2.1.7 Support packages

The math package now includes a fairly complete complex number class, and a fairly incomplete set of matrix, vector arithmetic, and signal processing classes.

The graph package contains an efficient graph representation and a small set of graph algorithms, with emphasis on utilities for operating on lattices and complete partial orders (CPOs), which are needed by the type system. The design emphasizes efficiency by representing the graph using an adjacency matrix, rather than a set of cross-referenced node and edge objects. But it also emphasizes convenient use by providing an interface in which arbitrary objects represent the nodes.

The plot package continues to improve slowly, although it is not a primary concern at this time since its current capabilities are adequate for our purposes.

2.1.8 User Interface

We had a number of frustrations in our effort to build a user interface for Ptolemy II, and as a net result, we have de-emphasized this part of the design. First, we lost confidence in Tcl/Tk as an infrastructure when Sun failed to support any object-oriented extension, and Itcl, the object-oriented extension that we were using, was late being updated to new versions of Tcl/Tk. We attempted a port of Tycho, our Itcl based UI toolkit, to Tcl++, but performance was abysmal. Second, our experiments with Swing, Sun's Java UI toolkit, indicated clearly that this technology had not matured. We decided to focus on having a good API for a UI in Ptolemy II, but to not put extensive resources into developing a graphical user interface as yet. Instead, we have focused on infrastructure for constructing applets that include Ptolemy II models.

We have made progress on defining the interface that a graphical user interface can use to interact with Ptolemy II. In particular, we have defined the following APIs:

System API. **Purpose:** Allow a UI to start up and connect to Ptolemy II. **Functionality:** Is Ptolemy installed?; query Ptolemy II version; query available domains; query actors fitting a given domain and name.

Execution API. **Purpose:** Allow UI to control a running simulation. **Functionality:** Start, pause, stop a simulation; notify on all execution events.

Icon Library API. **Purpose:** Allow UI to locate icons and related definitions. **Functionality:** What paths are searched; modify the search path; return top-level library names; return sub-library names; return file path of any library.

Domain API. **Purpose:** Allow UI to query a domain for parameters, display important run-time information (current time etc.). **Functionality:** What directors can be used; what information needs to be shown in the control panel; notify of time advanced.

Animation & Parameter APIs. **Purpose:** Allow UI to animate and display parameter of a running simulation. **Functionality:** Notify on: actor ready to fire; actor fired; actor blocked; token transmitted; token read.

Debug API. **Purpose:** Allow Tycho to act as a graphical "debugger" on a simulation **Functionality:** Step the simulation; set a firing breakpoint; set a value breakpoint; run to breakpoint; notify on breakpoint.

Logging API. **Purpose:** Allow logging and log-style debugging, with or without Tycho. **Functionality:** Notify on log message generated.

2.1.9 Mutable Systems

John Reekie defined a new "mutation" package for Ptolemy II designed to work with the Ptolemy II kernel to provide support for dynamically changing graph topologies. This package uses a publish-and-subscribe model, where the publisher informs subscribers of changes in the graph topology (mutations). It addresses the problem that we wish to constrain mutations to occur at precisely defined points in an execution cycle in order to ensure determinism.

This package includes an interface called Mutation with two methods, perform() and update(), that correspond to performing mutations and informing all observers about the mutations, respectively.

A second interface called MutationListener is implemented by any observer that depends on mutations. This interface has methods like addEntity, removeEntity, link, unlink, etc. corresponding to the elementary forms of mutations that are possible in the kernel.

Now any actor performing mutations would then create an object implementing the interface Mutation, and implement the methods perform() and update(). The perform method contains commands corresponding to all the mutations that the actor intends to make. The update() method specifies the same mutations but using the methods defined in the MutationListener interface.

After listing these methods and creating the Mutation object, the actor calls a method queueMutation() on the director and queues the mutation with the director. The director chooses when the mutations are actually performed, to ensure that occur at a safe point in the execution sequence.

This mechanism has been prototyped and demonstrated in the PN (process networks) domain.

2.2 TASK 2: DOMAIN-SPECIFIC DESIGN TOOLS

2.2.1 Process-Oriented Domains

We have created two domains where actors logically represent interacting processes. These are CSP (communicating sequential processes) [12] and PN (process networks) [14]. In CSP, communication occurs via rendezvous, where a single atomic action temporally links the two processes when communication occurs. In PN, communication occurs via asynchronous message passing, with FIFO queues serving as buffers. In the CSP domain, the emphasis is on controlled nondeterminsm, where for example a process can nondeterministically rendezvous with any of a set of other processes. In PN, the emphasis is on determinism, in that data sequences do not depend on scheduling details.

The main anticipated application for the CSP domain is for modeling resource management problems, such as embedded software multitasking and real-time scheduling. The anticipated application for PN is modeling real-time stream transforming processes and hardware subsystems.

In both cases, processes are implemented as Java threads. Mudit Goel and Neil Smyth created a subpackage of the actor package called actor.process specifically to hold the common infrastructure used by these two domains.

Both domains have been extended with timed versions, where communication events can be placed along a time line, and processes can request to be delayed by a specified amount of time. The semantics abstracts the passage of time in that it assumes that zero time passes between the external (communication or delay) events of an actor. That is, computation itself does not take time.

Both domains also support mutations, or changes in the graph topology that occur at run time.

2.2.2 Discrete-Event Domain

Lukito Muliadi has completed a prototype of the discrete-event domain in Ptolemy II. This domain uses a sophisticated calendar-queue scheduler to sort events, and is expected to play a major role in mixed signal design. Using this prototype domain, we have experimented with the first applet that intensively uses Ptolemy II infrastructure, a simple DE simulation that demonstrates the inspection paradox. This exercise help us verify that Ptolemy II infrastructure can be used effectively in applets.

2.2.3 Continuous-Time Modeling

Jie Liu has developed a continuous-time (CT) domain in Ptolemy II that includes a sophisticated, variable-step-size ODE solver, plus an ability to generate and to react to discrete events. Reaction to discrete events is accomplished by abstracting those events as Dirac delta functions. This domain is thus intended to interoperate with discrete domains such as DE and FSM.

2.2.4 State Machines and Hybrid Systems

During this reporting period, Xiaojun Liu made progress on defining an FSM domain in Ptolemy II. This domain will be used to specify control logic for modal models and for digital controllers. Specifically, he has extended Neil Smyth’s expression evaluator to support expressions that serve as guards for state transitions and actions associated with state transitions.

Meanwhile, Bilung Lee has implemented a valued FSM syntax in Ptolemy Classic. One key issue that he had to resolve was the need to be able to distinguish between the evaluation based on the status of events (i.e. whether events occur) vs. their values. For example, when a user writes “e1 || !e2”, does this means “e1 is present or e2 is not present” or “e1 is non-zero or e2 is zero”?

The approach he used is adapted from the Statemate. The transition of the FSM is denoted as

```
te [tc] / ta
```

where

- A trigger event “te” is a boolean expression generated by

  ```
  te ::= TRUE | FALSE | e | !te | te&&te | te||te
  ```

  where e is an input event, and the evaluation of an event e is either TRUE or FALSE when the event is either present or absent and "!", "&&" and "||" are the logic operators "not", "and" and "or", respectively.
- A trigger condition “tc” is a boolean expression generated by

  ```
  tc ::= nil | TRUE | FALSE | e | !tc | tc&&tc | tc||tc

                    | v==v | v!=v | v<v | v<=v
  ```

  where e is an input event, and the evaluation of an event e is either TRUE or FALSE when the value of the event is either non-zero or zero, and "!", "&&" and "||" are the same as those in trigger event te, and "==", "!=", "<" and "<=" are the logic operators "equal to", "not equal to", "less than" and "less than or equal to", respectively, and v is an arithmetic expression generated by

  ```
  v ::= e | c | v+v | v-v | v*v | v/v
  ```

  where e is an input event, and the evaluation of an event e in v is the value of that event. c is a constant value, and "+", "-", "*" and "/" are the arithmetic operators "add", "subtract", "multiply" and "divide", respectively.
- A trigger action “ta” lists a subset of the output events and is generated by

  ```
  ta ::= nil | e(v) | e | ta,ta
  ```

  where e is an output event. v is the same as that in trigger condition tc, and “,” distinguishes two events in the trigger action.

In one reaction of the FSM, a subset of the input events are present with some values. One transition is triggered if its guard is TRUE under the current input events. The guard consists of two parts, a trigger event and a trigger condition. If the trigger condition is null, it means that the trigger condition is omitted and then the guard is exactly the same as the one for pure FSMs. Otherwise, the guard is TRUE if both trigger event and trigger condition are TRUE. Then the FSM goes to the destination state of the triggered transition, and emits each output event in the trigger action of the triggered transition, making these output events present. If the output event to be emitted is attached with an arithmetic expression, the value of that event is set to be the evaluation of the expression. Otherwise, the value of that event is set to be 0. If the trigger action is null, it means that no output event is emitted.

2.2.5 Web-Based Simulation of Embedded Software (UT Austin)

Under subcontract, Brian Evans and his team at UT Austin have released a new version of their extensible, configurable, portable, freely distributable framework for Web-enabled simulation of embedded software for DSPs and microcontrollers. The new version, Version 1.0.6 (as of July 3, 1998), can be run by using a Java-enabled Web browser to open the URL

http://anchovy.ece.utexas.edu/~arifler/wetics/

Version 1.0.6 of the Web-enabled Simulation framework consists of:

- A set of Java applets that provide a configurable framework for Web-based user interfaces for instruction set architecture simulators.
- A multithreaded TCP/IP Internet server written as a Java application that provides the interface between the Java applets and the simulators.
- Command-line simulators/debuggers written in C/C++ for a. Texas Instruments TMS320C30 floating-point digital signal processor simulator

    b. Motorola MC68HC11 microcontroller simulator

    c. Motorola MC56811 fixed-point digital signal processing simulator

    d. Motorola MC56LC811 fixed-point digital signal processing board debugger

They provide pre-built binaries of the simulators and debuggers for Windows '95/NT and Solaris 2.5 machines.

New command-line tools for DSP and microcontroller processors and boards can be added to the framework by only providing data about the tools - no Java applet or application code changes. All of the data is kept in one file.

Their framework is in the spirit of the Web-based Electronic Design (WELD) Project at UC Berkeley directed by Richard Newton. WELD “aims to construct the first operational prototype of a national-scale CAD design environment enabling Internet-wide IC design for the U.S. electronics industry”. WELD focuses on VLSI CAD tools and design management infrastructure, and UT Austin's framework provides a complementary focus on embedded software.

2.2.6 Automated Multi-Criteria Filter Optimization Framework (UT Austin)

Under subcontract, Brian Evans and his team at UT Austin have released a set of Filter Optimization Packages for Matlab:

http://www.ece.utexas.edu/~bevans/projects/syn_filter_software.html

The packages to optimize the following characteristics of analog filter designs simultaneously

1. magnitude response
2. linear phase in the passband
3. peak overshoot in the step response
4. quality factors (Q)

subject to constraints on the same characteristics. This framework takes both behavioral and implementation characteristics of filters into account. A traditional approach has been to design a filter according to behavioral properties and then iteratively tweak the filter poles and zeros for a given implementation technology. This framework designs filters that simultaneously meet both behavioral and implementation constraints and goals.

2.3 TASK 3: HETEROGENEOUS INTERACTION SEMANTICS

2.3.1 Multi-Domain Modeling

Jie Liu and Lukito Muliadi created the first multi-domain simulation in Ptolemy II. This model has a CT (continuous-time) model inside a DE (discrete-event) model. This combination is particularly useful for mixed-signal modeling.

2.3.2 Time

We are gaining a much better understanding of the role of time in binding heterogeneous semantics. We have determined that continuous-time modeling fundamentally requires that either the discrete environment or the continuous-time modeling environment itself must be capable of rolling back a simulation. We are developing a framework in which it is sufficient for only the CT modeling environment to have this capability. This greatly simplifies the other modeling environments.

2.3.3 Data and Domain Polymorphism

In object-oriented languages, polymorphism is the independence of code from the data types it operates on. The same operation takes different forms depending on the data. This form of polymorphism, which we call *data polymorphism*, is supported in Ptolemy II in a number of ways. First, the token classes, which encapsulate data communicated between actors, polymorphically implement various arithmetic and logical operations. Thus, actors that require only these operations can easily be polymorphic in that they need not know *a-priori* what type of data they are operating on.

A second, more interesting kind of polymorphism is what we call *domain polymorphism*. Here, an actor does not know *a-priori* which domain it operates in. Thus, for example, although it assumes it will obtain input data, it does not know whether the input will be transferred to it by rendezvous, by asynchronous message passing, or by some other mechanism. We have begun the development of a set of domain-polymorphic actors, and in so doing, have been learning a great deal about the issues involved. One of these is dealt with in the next section.

2.3.4 Strictness

We have determined that we need a generalized notion of strictness for actors that is supported in the kernel. In programming languages, “strict” means that a function or procedure needs to have all its arguments in order to be able to calculate any output. For actors, the question is whether the inputs need to be known on all input ports in order for the actor to be able to produce outputs. Note that “knowing an input” means knowing the number of available tokens (which may zero) and knowing the values of the available tokens.

We have several uses for non-strict semantics. In the CT domain (continuous time), it is often important to be able to determine an estimate of an output value using an estimate of an input value. Thus, the value of the input tokens is not “known.” Only an estimate of it is known. It is important that when an estimate of the output is calculated, that the state of the actor not be updated. The state should be updated only when the input is completely known.

In synchronous/reactive modeling, for which we have not yet built a domain, it is essential to be able to assert outputs even when it is not known whether the inputs will have tokens. We identify three fundamental levels of strictness for actors:

- STRICT: An actor needs to know all inputs to perform its function.
- SOFT_STRICT: Given an estimate of all the inputs, the actor can produce estimates of the outputs.
- NON_STRICT: An actor can produce partial information about the output given partial information about the input. By “partial information” we mean that it need not be known whether an input has tokens, or how many tokens it has.

In both of the latter two cases, the actor should not update its state until the input is fully known. Typically, we will implement this by updating the state of the actor in its postfire method.

Domain-polymorphic actors (those that can operate in more than one domain) would need to assert which of the three levels of strictness they follow. Opaque composite actors, which mediate the interaction between domains, would defer to the inside director to determine strictness. Thus, a domain that is able to expose a NON_STRICT interface to other domains, for example, could assert NON_STRICT semantics.

The CT domain would prefer SOFT_STRICT actors, but would be able to handle STRICT actors by introducing a delta delay. Since CT is based on convergence to a fixed point at each time instant, it normally needs to invoke actors repeatedly with estimates of the inputs until convergence is reached. For STRICT actors, it would only invoke them once, after the inputs are fully known. At that time instant, the output is already known because of the delta delay, so convergence is not compromised.

Most domains embedded within CT, therefore, would have a delta delay from inputs to outputs. CT can also make use of NON_STRICT actors to overcome stiffness problems.

2.3.5 Interoperability

We have invested considerable effort into evaluating Sun's Java-based RMI (remote method invocation) vs. CORBA as technologies for multi-source distributed modeling and simulation. John Davis and Mudit Goel have prototyped small systems using both mechanisms, and have reached some tentative conclusions. RMI is much simpler than CORBA, but is limited to distributed applications where all components are written in Java. However, it is also apparently possible to use a subset of RMI to develop CORBA-compliant distributed applications. A talk at Java One (the major Java conference) included examples showing a simple distributed application with a mixture of Java RMI technology code and C++ CORBA code.

Jie Liu and William Wu prototyped a CORBA-based approach to reduced-order modeling in Ptolemy II. In this prototype, a CORBA server provided a reduced-order model of a MEMS accelerometer, while a CORBA client integrated the model within a larger model. The reduced-order model was provided by Dr. Per Ljung of Coyote Systems. The larger model was implemented in the CT domain of Ptolemy II. Eventually, we hope that this infrastructure can be used to solicit parameterized reduced-order models on demand from servers on the network.

CORBA is surprisingly complex, requiring considerable expertise to use. We are concerned that the cost and complexity of setting up CORBA ORBs may preclude many possible uses within the Composite CAD community.

We are also studying the use of JavaBeans with CORBA, and it appears that with CORBA 3.0, due out in the fourth quarter of this year, there could be a very useful synergy here. A 2/2/98 PC Week article, 'JavaBeans key to CORBA upgrade,' states:

> *"Skeptics have long complained that CORBA lacked the ease of use known to Microsoft Corp.'s rival COM (Component Object Model). But CORBA 3.0's JavaBeans object model could solve that. The new model, dubbed CORBAbeans, will enable rapid development of applications, with easier-to-use tools and interfaces."*

Also, Beans has an interface called InfoBus that allows Beans to communicate within the same virtual machine, something more difficult to accomplish with CORBA alone.

Finally, Christopher Hylands has investigated the Tcl Bean, which connects Tcl scripts via Java Beans to Java Studio, an environment distributed by Sun for graphically composing Java Beans. Christopher was studying how this might relate to Ptolemy II.

2.3.6 Design Flow Management

William Wu created an exploratory demo of a mechanism in Ptolemy II for managing design flows at a high level of abstraction. The purpose of this demo is to study the interaction between tools and block diagram semantics of Ptolemy. The scheduler in this demo is static, since scheduling is still an open issue. This work is not intended to turn directly into releasable code.

The flow in the demo performs the follow tasks

- Design a IIR Butterworth filter.
- Compute its frequency response and get postscript version of the plot.
- Filter a sample audio file through it.
- Compute the spectrogram of the original audio signal, and filtered signal and print postscript version of both spectrograms.
- Synthesize a Xilinx X4000 FPGA netlist.
- Display all three postscript plots in ghostview.
- Display the netlist in emacs.

These tasks mix Ptolemy package libraries with external tools, such as Matlab, ghostview, emacs, and the BOOM IIR generator for FPGAs from the BRASS project. The latter was wrapped in a CORBA wrapper. William, together with Jie Liu, writes this summary of the DFM effort:

> *"Traditional system-level design techniques usually rely on one particular well-defined model of computation (MoC) and try to map the application onto that model. As the size and scope of the system increase, no single MoC alone can manage the complexity. This forces the designer to leverage on the integration of multiple CAD tools. Incorporating different tools with simple scripts has become too difficult and unmanageable, since there is no standard defined interface between the tools. Ptolemy manages the interaction among different MoCs, and is capable of interoperating with external tools. Thus a design can be described by the MoCs supported within Ptolemy, and those used by external tools. For example, a system can be modeled partly in abstract block diagram using data flow model and partly in SPICE circuits. This heterogeneity exists not only in different MoCs, but also in different levels of abstractions of the design flow. The key is to manage the tools in the design flow and classify the tools by their semantics and their level of abstraction. Then the interoperability of the tools becomes the interoperability of the semantics. Ptolemy can act as the standard interface among different tools to achieve the overall design requirements. Three types of semantics/tools interaction are studied. They are domains within Ptolemy, local interprocess communication, and global object retrieve and invocation through CORBA."*

2.4 GENERAL INFRASTRUCTURE

We have done quite a bit of work that contributes to each of the three tasks without being specifically part of any one of them. This section summarizes that work.

2.4.1 Software Practice

One of the major innovations in the Ptolemy project is the development of a practical, usable, and systematic software engineering practice in an academic setting. This experiment has been masterminded by John Reekie [52].

In this practice, each class is advanced through four levels of confidence by a light-weight review and testing process. There are two ratings for each class: that proposed by its author or maintainer, and that accepted by its tester and reviewers. This approach tries to maintain the accepted principles of code review and testing by people other than the author, while keeping overhead manageable for a research group and not squelching innovation.

The basic idea is that the author proposes that a class advance a level. The tester/reviewer is then responsible for examining the class (with the help of other reviewers, if necessary), writing test code for it, and either accepting or rejecting the proposed advancement. The tester/reviewer needs to provide specific and concrete reasons for rejection; the author is obliged to make needed modifications and re-submit the code.

The four levels of confidence are red, yellow, green, and blue. Each level has a well-defined meaning, which both the author and tester/reviewer are expected to satisfy before proposing or accepting advancement to that level.

The process includes both design reviews and code reviews. The four ratings are as follows:

Red. All code starts here. Red code is in flux, and anyone that calls red code should expect it to change without warning. Code that calls red code does not need to be modified by the author if changes in the red code break the calling code. Red code should never be released, in theory, although in practice, we do release it with appropriate caveats.

Yellow. The interface and overall design of the class is acceptable for development purposes. Clients can code to this interface with the expectation that further changes will be limited to revisions, not major changes. Clients cannot insist that code that calls yellow code will work at all, in that all that has to be defined is the interface, not the implementation. With this view, Yellow code should generally not be released. In practice, we have rarely promoted code to yellow without the interface being backed by a complete implementation, and hence we are reasonably comfortable releasing yellow code.

Proposing advancement. The author proposes advancement to yellow when he/she is satisfied with the design of the class, and how it collaborates with other classes. The author is responsible for making sure that the class is adequately documented. UML diagrams at the design level are appropriate here.

Accepting advancement. The tester/reviewer accepts advancement to yellow when he/she is satisfied that the design of the class is satisfactory in the context given by the author. The tester/reviewer should evaluate the class solely on the basis of the documented interface -- note that, in the case of classes which are designed to be subclassed, this includes the (protected) interface provided to subclasses. The tester/reviewer is entitled to request UML diagrams as an aid to understanding the purpose and function of the class, and is entitled to organize a design review at this time. The tester/reviewer is also entitled to require changes to the interface in anticipation of testing needs.

Acceptable changes. Yellow code can have interface changes before advancing to green, but the author should avoid making wholesale changes. If wholesale changes are required, the author should request that the class be taken back to red. If the author changes the interface to yellow code, then he is responsible for a) making sure that the calling code compiles, and b) notifying the author of calling code about the change. The author is not responsible for making sure that the calling code works or passes its test suites, as yellow code does not provide any assurance of functionality.

Green. The interface of the class has been finalized, and the implementation is acceptable for development purposes. The documentation of the interface is acceptable for development purposes. Clients can code to green code in the expectation that the interface will not change in such a way as to break compilation, nor will the implementation change enough to break the caller’s test suite. Green code can be released.

Proposing advancement. The author proposes advancement to green when he/she is satisfied that the interface to the class will not need further changes, except for relatively minor enhancements and additions, and with the implementation of the class. This could be considered "beta" level code. The author is responsible for providing the tester/reviewer with a test suite that he/she can use as a starting point. This test suite should exercise and illustrate the main uses of the class.

Accepting advancement. The tester/reviewer accepts advancement to green if he/she is satisfied with the implementation of the class. The tester/reviewer shall determine this by a) writing a test suite, and b) reviewing the code. The test suite must have at least 50% code coverage. The tester/reviewer is entitled to organize a code review at this time. The author is required to write example test code if requested by the tester/reviewer.

Acceptable changes. The interface to green code can have interface changes, but these should be minor or purely incremental (new methods). If the changes break any test suites, the author is required to fix the calling code. If a change will require substantial fixes, then the class should probably be taken back to yellow.

Blue. The implementation of the class has been fully and completely tested, and accepted as meeting all requirements. All documentation, including external documentation if appropriate, is complete. Blue code can be released. We have not yet advanced any Ptolemy II code to blue, although by these criteria, the kernel and actor packages could probably be advanced to blue at this time.

Proposing advancement. The author proposes advancement to blue when he/she is satisfied that the class is finished, polished, flexible, and robust. This applies to the documentation as well as the code. This is, in other words, quality releasable code. In general, code should not be advanced to blue until it has been in use by other classes for some time.

Accepting advancement. The tester/reviewer accepts advancement to blue when he/she is satisfied that the class is finished, polished, flexible, and robust. The tester/reviewer determines this by completing the test suite to get 100% coverage, or as close as is reasonable given the way that the class operates with other classes, and possibly also by writing a test suite that tests this class working in collaboration with other classes.

Acceptable changes. Blue code can have bug fixes, but changes to the external interface, inherited interface, or observable behavior, make the class a candidate for reversion to green or yellow status.

John Reekie conducted a design review of our review process. This “meta-review” or “review review” yielded some interesting insights that have led to some fine tuning of the process.

2.4.2 Support Software

In a collaboration between the Ptolemy project and the CAD group at Berkeley, John Reekie and Michael Shilman have developed a Java package called Diva, which is a first step toward a software infrastructure for visualizing and interacting with dynamic information spaces. Diva’s distinguishing characteristics are its emphasis on dynamic data and interactive user interfaces, and its clean, easy-to-use API.

The most recent release consists of two Java packages, diva.canvas and diva.graph. The Diva canvas is a structured graphics layer over the Java2D API with a coherent set of features intended to support innovative and interesting visualizations and user interfaces. The Diva graph package implements an easy-to-use architecture for graph visualization, based on a Swing-style MVC data and notification model. Extensible and pluggable graph rendering and layout facilities complete the picture.

2.4.3 *Other Software*

Under the structure of a weekly study group, we have made systematic studies, including prototyped experimental implementations, of each of the following technologies:

- CORBA
- Java RMI
- JINI
- JavaSpaces
- VHDL and Verilog

We have also performed experiments with the following software packages:

- Rational's Quantify for Java
- Optimizit, a commercial product for profiling Java software
- Saber (from Analogy)

3. Software

Software in the Ptolemy project serves as both a laboratory for experimentation and a mechanism for disseminating results. A new feature of this project is that we expect to be distributing software in the form of smaller packages rather than large monolithic software systems. During the second year of the project we completed several small software releases.

3.1 INFORMATION DISSEMINATION POLICY

The Ptolemy web site, http://ptolemy.eecs.berkeley.edu, is used to distribute all software (including source code) and documentation, together with updated summary sheets, answers to frequently asked questions, and tutorials. We use the most liberal copyright permitted by the University of California, one which has proven effective in promoting technology transfer. A Usenet news group called comp.soft-sys.ptolemy and a mailing list ptolemy-hackers@ptolemy.eecs.berkeley.edu are used to communicate with outside users. Postings to the mailing list are cross-posted to the news group. Postings are archived and searchable from our web site.

3.2 DEVELOPMENT ENVIRONMENT

During this reporting period, we switched from SCCS to CVS as our primary version control infrastructure. This coincided with a much greater emphasis on the use of laptop computers running Windows NT in the development environment. Christopher created a package (a self-extracting archive) that includes CVS and SSH (a secure shell that CVS uses to communicate without sending cleartext passwords).

We have also developed an extensive regression testing infrastructure for Ptolemy II. The tests are written in Tcl, and Jacl (a pure Java implementation of Tcl) is used to execute the Tcl. Christopher Hylands and John Reekie worked with Mo DeJong (University of Minnesota) and Bryan Surles (Scriptics Inc.) on Tcl Blend and Jacl.

3.3 SOFTWARE RELEASES

We released the first version of Ptolemy II, designated 0.1alpha, in December 1998. This release includes the kernel, actor, and data packages, but none of the domains. The release is intended for collaborators and critics of the software architecture.

We released Ptplot 2.0 in December 1998 and Ptplot1.3p1 in June 1998. Ptplot is a Java 2-D plotting package that can be used as an applet or a stand-alone application.

We released Ptolemy 0.7.1, the most recent version of Ptolemy Classic, in alpha and beta versions in May, 1998, and in final version in June, 1998. While much of Ptolemy 0.7.1 is independent of this HMAD project, it has the following connections:

- Mutable discrete-event modeling
- Starcharts generalized hybrid systems
- The NT port (which gives us experience developing and releasing NT software)
- Ptplot (which gives us experience developing and releasing Java software)

4. Plans for the next year

Capabilities that we anticipate making available in the next year include:

- *Extensible XML-based file formats*. XML is an emerging standard for representation of information that focuses on the logical relationships between pieces of information. Human-readable representations are generated with the help of style sheets. Ptolemy II will use XML as its primary format for persistent design data.
- *Formalization of process-level type systems*. We plan to use the development of a domain-polymorphic actor library to drive better understanding of process-level types. Process-level types capture dynamic properties of the interaction between simultaneously active objects.
- *Interoperability through software components.* Ptolemy II will use distributed software component technology such as CORBA, Java RMI, or COM, in a number of ways. Components (actors) in a Ptolemy II model will be implementable on a remote server. Also, components may be parameterized where parameter values are supplied by a server (this mechanism supports *reduced-order modeling*, where the model is provided by the server). Ptolemy II models will be exported via a server. And finally, Ptolemy II will support migrating software components.
- *Embedded software synthesis.* Pertinent Ptolemy II domains will be tuned to run on a Java virtual machine on an embedded CPU. Hardware, firmware, and configurable hardware components will expose abstractions to this Java software that obey the model of computation of the pertinent domain. Java's native code interface will be used to define a stub for the embedded hardware components so that they are indistinguishable from any other Java thread within the model of computation. Domains that seem particularly well suited to this approach include PN and CSP.
- *Embedded hardware synthesis*. Earlier versions of Ptolemy had only very weak mechanisms for migrating designs from idealized floating-point simulations through fixed-point simulations to embedded software, FPGA, and hardware designs. Ptolemy II will separate the interface definition of component blocks from their implementation, allowing libraries to be constructed where compatibility across implementation technologies is assured. This work is currently being prototyped in Ptolemy 0.7.1.
- *Integrated verification tools.* Modern verification tools based on model checking 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.

5. Technology Transfer

One of the notable properties of the Ptolemy project is its track record of demonstrable transfer of technology to industry leaders in the computer-aided design and defense industries. This is accomplished via a careful cultivation of industry contacts and a strategy of wide open, very liberal distribution of software and publications. All software is made available on the Web with the most liberal copyright notice permitted by the University of California. This notice retains ownership of the copyright, but expressly grants permission to use the software for any purpose, including development of commercial products. It is distinctly more liberal than the GNU public license, and thus better represents "free software."

Although it is still too early in this project for major results of the project to have been transferred, there are some ongoing interactions that we wish to highlight.

5.1 HEWLETT-PACKARD

HP has released a beta version of HP Ptolemy in January 1998, and a final version shortly thereafter. HP Ptolemy is based on Ptolemy Classic. Some of the major additions over the Berkeley Ptolemy code are:

- Runs On NT 4.0, NT 3.51 and Win 95
- Added time and frequency simulation and modeling
- Added DSP filter tool
- Added VHDL modeling and simulation
- Added Verilog modeling and simulation
- Added 300-400 time and frequency models
- Added Spice and Harmonic Balance cosimulation....

HP has integrated their HF Spice, Harmonic Balance and Circuit Envelope simulators into HP Ptolemy. Their simulation executable links two distinct simulator software architectures: one based on UCB Ptolemy (HP Ptolemy) and the other incorporating the HP EEsof analog simulators (Gemini). In HP Ptolemy there currently are two simulation domains: SDF and TSDF. In TSDF, HP has modified the semantics of SDF to introduce a notion of time to apparently make it easier to combine DSP simulations with analog simulations. Using the TSDF domain a user can embed a circuit simulator into a dataflow simulation using an interface very similar to a Ptolemy wormhole.

At the DSP Spring conference in Santa Clara on April 13, 1998, Hewlett-Packard announced a complete HP W-CDMA Design Library based on HP Ptolemy. According to Paul Washkewicz of HP,

> *"This is one of the hottest technologies to hit the streets recently, and our ability to get this product into the industry is directly related to our adoption of UCB Ptolemy."*

Paul also told us privately that

> *"With HP Ptolemy as our framework, we fully expect to deliver almost a new product every month for the next 6 months. This is the real value in HP Ptolemy."*

5.2 NASA

Ivan Clark of NASA contacted us and told us that "NASA is involved in a research program that uses our Ptolemy software as part of a modeling effort. As part of the NASA aviation programs, there has been underway for 2 or 3 years an effort to develop an Integrated Electromagnetic Sensor Simulation (IESS). growing out of previous Fortran-based modeling efforts to simulate windshear detection by radar, the IESS program has been trying to modularize and convert the fortran code into Ptolemy-compatible modules and to add air-to-air multiple radar-target tracking to the simulation. the next generation of the model effort will seek to add the capability of simulating the flying of sensors (radar, lidar, radiometer, etc.) through a weather model with emphasis on detection of aviation weather hazards such as turbulence. this next-generation effort currently includes participation by NASA, RTI, NCAR, Allied Signal, and Collins."

5.3 CADENCE

On Oct. 20, 1998, we met with 8 engineers and managers from Cadence Design Systems at Berkeley for half a day to brief them on our approach to integrating control logic with dataflow-oriented modeling. Cadence has commercialized techniques from the Ptolemy project in the past, although during this reporting period there were no new announcements.

5.4 BNED / VIRTUAL PHOTONICS

BNeD, now merged with Virtual Photonics, has announced the next generation product line, the "Photonic Transmission Design Suite" (PTDS), which is partly based on Ptolemy. They say, "The PTDS features a modular approach to optics and photonics simulation, offering distinct libraries for component, system and network design together with a unique and innovative simulation architecture."

BNeD and Hewlett-Packard have announced a joint marketing agreement to distribute their simulation environment for optical fiber communication systems. Wolfgang Reimer, who has done a lot of work with Ptolemy and Linux, is part of BNeD. He visited us on Feb. 27 with Igor Koltchanov, the BroadNeD Product Manager, and demonstrated the simulation environment, which focuses on the physical properties and their interaction with communication algorithms. The fact that HP's new design environment is based on Ptolemy made this marketing agreement a clear win for both sides.

5.5 PHILIPS

Three visitors from the Philips Research Lab in Eindhoven, The Netherlands, Bart Kienhuis, Kees Vissers, and Pieter van der Wolf visited our group on March 13 and presented a model of computation that they are using for embedded video systems that is heavily inspired by our own work on Kahn process networks. Their model of computation blends Kahn process networks with a model called PAMELA that adds time. This notion of time should make it much easier for us to mix PN models with physical models where time is intrinsic.

Stimulated by this visit, Mudit Goel spent the summer at Philips Research Labs in Eindhoven. The appendix includes a report from Pieter van der Wolf of Philips. Mudit was working on embedded system design methodologies, including the modeling of applications and programmable architectures for real-time video.

5.6 CADABRA

Farhana Sheikh, a Project Leader at Cadabra (http://www.cadabratech.com) reports that they used Tycho (our Itcl-based UI toolkit) to generate the C++ class diagram for their physical synthesis software. The software contains a little over 700,000 lines of C++. She reports that parsing the code did not take very long. However, it was a bit of a struggle to get it all printed out on our plotter. They now have an entire wall covered with our class hierarchy which has proven useful in training new developers.

5.7 THE MATHWORKS

Prof. Lee visited the MathWorks, makers of Matlab and Simulink, on June 2, 1998, to discuss the trajectory of ongoing work with Simulink and Stateflow. Approximately 20 engineers and managers, including the company president, were present for the meeting, which lasted all afternoon. We offered a critique of the current discrete-time model in Simulink and of the interaction between Simulink and Stateflow, and received in return an advance view of forthcoming product offerings. Simulink and Stateflow work together on a principle that is a special case of our *charts (starcharts) model of computation.

5.8 COYOTE SYSTEMS

Dr. Per Ljung of Coyote Systems provided us with two Simulink models that are a manually-modified automatically generated model of a MEMS accelerometer. The first is a simple 2nd order system of the proof mass (no damping), and the second is a comb finger with sensing of position. Applying a square wave excitation results in deflection of the proof mass and a capacitance change in the comb finger sensor. This is graphically shown in the output of the simulink simulation. Jie Liu successfully converted these models to Ptolemy II under the CT domain.

5.9 TECHNOLOGIES LYRE INC.,

Technologies Lyre Inc., of Quebec, Canada, demonstrated in their booth at DSP World Spring, Santa Clara CA, April 21-23, a rapid-prototyping platform called SignalMaster, based on Motorola's DSP56301 as well as a newer SHARC-based model. They are offering Ptolemy Classic as one of the main interfaces bundled with their DSP boards. They use the code generation capability to produce downloadable code. On May 6, we met with two representatives of Lyre at Berkeley to discuss whether we would participate in a beta test of their hardware. We decided not to.

5.10 IMPROV SYSTEMS

Cary Ussery <caryu@improvsys.com>, from Improv Systems Inc., visited our group on April 24, and discussed their software and hardware design methodology. Improv Systems is a startup company (started this past August) which has developed a new processing platform for embedded systems.

5.11 TCL/TK & JAVA TUTORIAL

Christopher Hylands and John Reekie presented a tutorial, "Tcl and Java Programming: Practice and Pitfalls," on September 15th, 1998 at the 1998 Tcl/Tk workshop, San Diego. There were about 45 attendees. Tutorial notes including slides are at:

http://ptolemy.eecs.berkeley.edu/~johnr/tutorials/tcljava98/

5.12 COOPERATION WITH OTHER GROUPS AT BERKELEY

We have begun a collaboration with the group of Prof. Kris Pister, who is working on MEMS-based semi-autonomous microrobotic agents. Members of his group have used Ptolemy II (the DE domain) to construct simulations of multiple interacting agents.

The POLIS team at Berkeley and Cadence (headed by Professor Alberto Sangionvanni-Vincentelli) has developed POLIS, a co-design environment for control-dominated embedded systems. POLIS, which is based on Ptolemy Classic, using primarily the DE domain, offers an integrated interactive environment for specification, co-simulation, formal verification, and synthesis of embedded systems implemented as a mix of hardware and software components.

The group of Prof. Dave Messerschmitt at Berkeley has developed the SiP Protocol Modeling tool that uses SPIN and the DE domain of Ptolemy Classic. Here is a summary:

> *“We investigated a widely distributed software package, SPIN, that supports the formal verification of distributed systems. PROMELA (Process Meta Language) is the input language of SPIN, which has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms. SPIN checks for the logical consistency of a specification, deadlocks, race conditions, etc.”*

5.13 COOPERATION WITH OTHER UNIVERSITIES

David V. Anderson (dva@ee.gatech.edu), of the Center for Signal & Image Processing at the Georgia Institute of Technology has informed us of plans to significantly extend our Ptolemy signal plotter. We are remaining in touch, and providing support as needed.

Prof. Brian Evans and some of his students visited a number of times. They met with several group members to help coordinate our ongoing collaboration (they are subcontractors on this project).

Neil Smyth spent one month at Delft University in The Netherlands. He was working closely with Bart Kienhuis on his simulator for stream based function (SBF) objects. In particular Neil added the notion of time to his simulator, with the intent of giving feedback earlier in the design cycle to the designer on issues such as processor utilization and bottlenecks. Neil says, “One of the main advantages for me of my stay at Delft University was seeing the application of much of the theory I have read about to practical problems.”

Frédéric Boulanger, of Supélec in France has developed a code generation domain for SR (our synchronous/reactive domain in Ptolemy) named SRCGC. Code generation for pure SR systems currently works, and they are studying code generation for mixed domain systems. Their final goal is to provide tools for separate development of the control and data processing parts of an application. SR and SDF (synchronous dataflow) could be used for prototyping, and SRCGC and CGC (code generation in C, which uses dataflow semantics) for the production of draft code, the final application code could require a last manual step before meeting size and speed requirements. In the process, Boulanger has provided a number of improvements to the SR implementation, and we have agreed that he should, in the near term, become the primary maintainer of this code. Boulanger has also made a new version of ocpl, which translates oc modules (from Lustre or Esterel programs) into Ptolemy blocks in the DE, SDF, SR, CGC and SRCGC domains. This work is done in collaboration with Xavier Warzee at Thomson TCO, and they agreed to make the new domain publicly available.

The Universitat de Girona is studying the Ptolemy environment, and using it to make significant steps into DSP development.

Jens Voigt of the Technical University in Dresden, Germany, has contributed a new mechanism in the Ptolemy discrete-event (DE) domain in Ptolemy Classic that he calls “Dynamic Higher-Order Functions”. This mechanism supports dynamically mutating topologies. Jens has applied this to the simulation of wireless radio applications. John Davis, in the Ptolemy group, has been working with Jens to integrate and extend his code. For example, included in his mechanism are “DEDynamicFork” and “DEDynamicMerge” stars, which can vary the number of portholes during runtime. Each time he instantiates one more block in the wireless radio simulation, he adds one porthole to a multiporthole in these dynamic stars. Ptolemy II will support such mutability in a robust and complete way.

6. Acknowledgments

6.1 PARTICIPANTS AT BERKELEY

6.1.1 Principal investigator

- Edward A. Lee

6.1.2 Professional staff

- Christopher Hylands
- Jennifer Basler
- Mary Stewart

6.1.3 Post-doctoral researchers

- James Lundblad
- John Reekie

6.1.4 Graduate students

- John Davis, II
- Stephen Edwards
- Ron Galicia
- Mudit Goel
- Bilung Lee
- Jie Liu
- Xiaojun Liu
- Steve Neuendorffer
- Neil Smyth
- Jeffrey Tsay
- William Wu
- Yuhong Xiong

6.1.5 Undergraduate students

- Michael Leung

6.2 CORPORATE SUPPORT

6.2.1 Sponsors

The following organizations have contributed additional financial support for the Ptolemy project:

- the State of California MICRO program
- Cadence Design Systems
- Hewlett-Packard
- Hitachi
- Hughes
- Motorola

- NEC
- Philips
- the Semiconductor Research Corporation (SRC)

7. Publications

Following is a list of publications with significant content developed under this project.

7.1 JOURNAL ARTICLES

[1] 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, to appear, 1998. Also UCB/ERL Memorandum M98/7, March 4th 1998.

[2] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,” to appear, *IEEE Transactions on CAD*, (Revised from ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997).

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “Synthesis of Embedded Software from Synchronous Dataflow Specifications,” Invited paper, to appear in *J. of VLSI Signal Processing*, 1998.

7.2 CONFERENCE PAPERS

[4] 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.

[5] H. J. Reekie and E. A. Lee, “The Tycho Slate: Complex Drawing and Editing in Tcl/Tk,” April 27, 1998. *Proc. Sixth Annual Tcl/Tk Conference*, September 14-18, 1998, San Diego, California.

[6] 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.

7.3 TECHNICAL REPORTS

[7] E. A. Lee, “Overview of the Ptolemy Project,” ERL Technical Report UCB/ERL No. M98/71, University of California, Berkeley, CA 94720, November 23, 1998.

[8] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, “Heterogeneous Concurrent Modeling and Design in Java,” Technical Report UCB/ERL No. M98/72, University of California, Berkeley, CA 94720, November 23, 1998.

[9] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, “Resynchronization for multiprocessor DSP implementation - part 1: Maximum-throughput resynchronization,” Tech. Rep., Digital Signal Processing Laboratory, University of Maryland, College Park, July 1998. Revised from Memorandum UCB/ERL 96/55, Electronics Research Laboratory, University of California at Berkeley, October, 1996.

[10] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, “Resynchronization for multiprocessor DSP implementation - part 2: Latency-constrained resynchronization,” Tech. Rep., Digital Signal Processing Laboratory, University of Maryland, College Park, July 1998. Revised from Memorandum UCB/ERL 96/56, Electronics Research Laboratory, University of California at Berkeley, October, 1996.

[11] 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).

7.4 MASTERS REPORTS

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

[13] 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.

[14] 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.

7.5 PHD THESES

[15] M. Goodwin, *Adaptive Signal Models: Theory, Algorithms, and Audio Applications*, Ph.D. thesis, University of California, Berkeley, December 1997. Available as UCB/ERL M97/31.

[16] M. Williamson, *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*, Ph.D. thesis, Memorandum UCB/ERL M98/45, Electronics Research Laboratory, University of California, Berkeley, May, 1998.

7.6 PUBLICATIONS PRODUCED UNDER SUBCONTRACT AT UT AUSTIN

The following publications were produced under subcontract to UT Austin:

[17] B. Lu, D. Wei, B. L. Evans, and A. C. Bovik, ``Improved Matrix Pencil Methods", Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers, Nov. 1-4, 1998, Pacific Grove, CA. http://www.ece.utexas.edu/~bevans/papers/1998/estimation/

[18] G. E. Allen, D. C. Schanbacher, and B. L. Evans, "Real-Time Sonar Beamforming on a Unix workstation Using Process Networks and POSIX Pthreads," Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers, Nov. 1-4, 1998, Pacific Grove, CA. http://www.ece.utexas.edu/~bevans/papers/1998/beamforming/index.html.

[19] S. Gummadi and B. L. Evans, "Cochannel Signal Separation in Fading Channels Using a Modified Constant Modulus Array," Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers, Nov. 1-4, 1998, Pacific Grove, CA. http://www.ece.utexas.edu/~bevans/papers/1998/constant_modulus/index.html.

[20] R. Bhargava, R. Radhakrishnan, B. L. Evans, and L. K. John, "Evaluating MMX Technology Using DSP and Multimedia Applications," Proc. IEEE Int. Sym. on Microarchitecture, Dallas, TX, Nov. 30 - Dec. 2, 1998. http://www.ece.utexas.edu/~bevans/papers/1998/mmx-pentiumII/index.html.

[21] B. Lu and B. L. Evans, ``Channel Equalization Performance of Two Feedforward Neural Networks," Proc. IEEE Conf. on Acoustics, Speech, and Signal Processing, Mar. 14-19, 1999, submitted.

[22] N. Damera-Venkata and B. L. Evans, ‘‘Optimal Design of Real and Complex Minimum Phase Digital FIR Filters,’’ Proc. IEEE Conf. on Acoustics, Speech, and Signal Processing, Mar. 14-19, 1999, submitted.

[23] B. Lu and B. L. Evans, ‘‘Low-Complexity Channel Equalization by Cascading Two Feedforward Neural Networks,’’ Proc. IEEE Int. Sym. on Circuits and Systems, May 30-Jun. 2, 1999, submitted.

[24] D. V. Tosic, M. D. Lutovac, and B. L. Evans, ‘‘Advanced Digital IIR Filter Design,’’ Proc. IEEE Int. Sym. on Circuits and Systems, May 30-Jun. 2, 1999, submitted.

[25] N. Damera-Venkata and B. L. Evans, ‘‘An Automated Framework for Multi-criteria Optimization of Analog Filter Designs,’’ IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, submitted.

[26] N. Damera-Venkata, B. L. Evans, and S. R. McCaslin, ‘‘Design of Optimal Minimum Phase Digital FIR Filters Using Discrete Hilbert Transforms,’’ IEEE Transactions on Signal Processing, submitted.

8. References

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

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

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

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

[37] 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.

[38] 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.

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

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

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

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

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

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

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

[46] 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.

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

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

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

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

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

[52] 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.

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

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

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

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