To appear in: Formal Methods and Models for System Design, edited by Sandeep Shukla and Jean-Pierre Talpin, Kluwer, 2004.

Chapter 2

ACTOR-ORIENTED MODELS FOR CODESIGN

Balancing Re-Use and Performance

Edward A. Lee
University of California at Berkeley
eal@eecs.berkeley.edu

Stephen Neuendorffer
University of California at Berkeley
neuendor@eecs.berkeley.edu

Abstract: Most current hardware engineering practice is deeply rooted in discrete-event modeling and synchronous design. Most current software engineering is deeply rooted in procedural abstractions. The latter says little about concurrency and temporal properties, whereas the former lacks many of modularity capabilities of modern programming languages. Actor-oriented design emphasizes concurrency and communication between components while maintaining modularity. Components called actors execute and communicate with other actors. In contrast to the interfaces in object-oriented design (methods, principally, which mediate transfer of the locus of control), interfaces in actor-oriented design (which we call ports) mediate communication. But the communication is not assumed to involve a transfer of control. This paper discusses the structure of actor-oriented models and shows how data and behavioral type systems enhance modularity and re-use potential while enabling designs that embrace concurrency and time. This paper shows how components can be designed for re-use through parameterization and behavioral polymorphism, and how component specialization can offset the performance costs of doing so.

Keywords: Actor-oriented design, behavioral types, behavioral polymorphism, component specialization, code generation, parameterization, Ptolemy.

1 INTRODUCTION

Current engineering practice for the codesign of embedded hardware and software systems draws heavily on the existing technology in these fields. For hardware design, register-transfer level (RTL) descriptions combined with a variety of optimization and transformation techniques have been effectively leveraged to design ASIC and FPGA systems. For software systems, high-level procedural languages, combined with sophisticated architecture-specific compilers have been leveraged to program a wide variety of microprocessors. However, while these approaches have flourished separately, there has been relatively little bridging the gap between the hardware and software design worlds.

One difficulty that both hardware designers and software designers share is the problem of constructing large designs with high-level abstractions. Both hardware and software languages have developed abstraction mechanisms based on the principles of data hiding and encapsulation, enabling design reuse. In software, significant progress has been made extending procedural languages with *objects* that associate data with a set of *methods* that access that data. Languages for building *models* of object relationships, such as the Unified Modeling Language and *design patterns* of common relationship structures, have further organized object-oriented design techniques. On the hardware side, *modules* are often used to hide the details of complex logic blocks. The internal logic of a module can only be accessed through externally visible *signals. Communication protocols* allow modules to declare high-level communication patterns and for those interfaces to be automatically replaced with the correct signals and control logic.

We claim that one of the reasons that technologies for hardware design and software design have remained separate is the vast difference between methods and signals. In particular, a method represents a *transfer of control*, forcing sequential execution of code. This sequential transfer of control manifests itself as the possibility of deadlock when concurrent threads execute the same piece of code. In contrast, signals represent pure *dataflow* between modules, allowing modules to operate concurrently. Because of the inherently sequential nature of object interaction, the concurrency of circuits, real-time software, and distributed systems-of-systems are difficult to represent in an object-oriented system.

Another significant difference between technology for hardware design and software design concerns *static structure*. In object-oriented languages, objects and object structures are dynamically created during execution. In most hardware design languages, however, the relationships between modules are fixed and do not change at run-time. Static relationships between modules are a significant aid for circuit synthesis, but stand in the way of using hardware design techniques for software and even complicate the design of systems with reconfigurable logic.

In this chapter, we describe a style of system modeling that is capable of expressing systems that are both concurrent (such as logic circuits and distributed systems), and also dynamically modifiable at run-time (as in software or reconfigurable logic). We call this style *actor-oriented design*. Actor-oriented models concentrate on the dataflow through a system and are thus able to elegantly describe concurrent systems. The interaction between actor-oriented components is described using well-understood patterns of interaction, called *models of computation*. Models of computation play the same role in actor-oriented models as design patterns in object-oriented languages and communication protocols in hardware design languages. Actor-oriented models can also be reconfigured at different levels of granularity. In order to soundly compose components and to provide for synthesis of hardware and software for these models, we describe several techniques for analyzing the static structure of these models.

2 ACTOR-ORIENTED DESIGN

A recent thrust in embedded systems research has been to develop domain-specific languages and synthesis tools for those languages. For example, Simulink, from The MathWorks, was originally created for control system modeling and design, and has recently come into significant use in embedded software development (using Real-Time Workshop and related products). Simulink is one of the most successful instances of *model-based design* [28]. It provides an appropriate and useful abstraction of control systems for control engineers.

Simulink also represents an instance of what we call *actor-oriented design*. Actors are concurrent components that communicate through *ports* and interact according to a common pattern of interaction, or *model of computation*. Primarily, actor-oriented design allows designers to consider the interaction between components distinctly from the specification of component behavior. Actor-oriented design allows us to compose objects with simple behavior to create more complex behaviors and to perform such compositions repeatably and robustly.

Our notion of actor-oriented design is related to the work of Gul Agha and others. The term “actor” was introduced in the 1970’s by Carl Hewitt of MIT to describe the concept of autonomous reasoning agents [8]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1]. Agha’s actors each have an independent thread of control and communicate via asynchronous message passing. We additionally embrace a larger family of models of concurrency that are often more constrained than general message passing. Our actors are still conceptually concurrent, but unlike Agha’s actors, they need not have their own thread of control. Moreover, although communication is still through some form of message passing, it need not be strictly asynchronous.

2.1 Actor-oriented models

In actor-oriented design, components called actors execute and communicate with other actors in a model. Actors have a well-defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes ports that represent points of communication for an actor, and parameters that are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed. The configuration of a model also contains explicit communication channels that pass data from one port to another. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

Like actors, which have a well-defined external interface, *models* (which are compositions of interconnected actors) may also define an external interface. External interfaces allow for *hierarchical abstraction*, illustrated in Fig.1. This interface consists of external ports and external parameters, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that comprise the model.

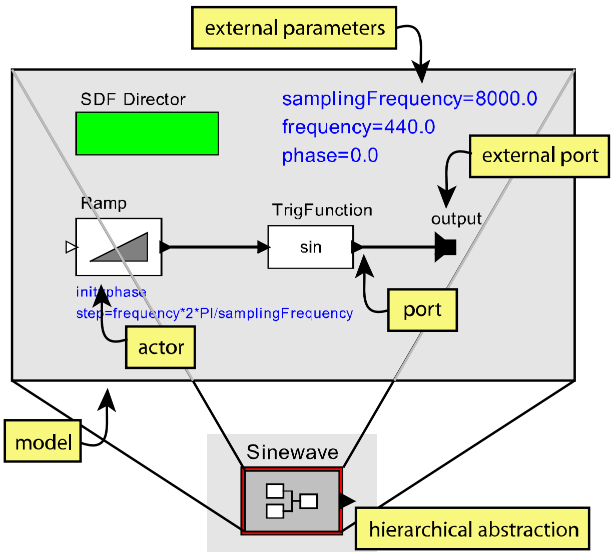

Figure 1: Hierarchical abstraction in actor-oriented design.

External parameters of a model can be used to determine the values of the parameters of actors inside the model.

Taken together, the concepts of models, actors, ports, parameters and channels describe the abstract syntax of actor-oriented design. This syntax can be represented concretely in several ways, such as graphically, as in Fig.1, in XML, or in a program designed to a specific API. Ptolemy II [18] offers all three alternatives.

In addition to Simulink, there are many examples of actor-oriented languages, frameworks, and software techniques, including Labview (National Instruments), Modelica (Linkoping), GME: generic modeling environment (Vanderbilt) [12], Easy5 (Boeing), SPW: signal processing worksystem (Cadence), System studio (Synopsys), ROOM: real-time object-oriented modeling (Rational) [26], VHDL, Verilog, SystemC (Various), Polis & Metropolis (UC Berkeley) [6], and Ptolemy & Ptolemy II (UC Berkeley) [18].

Many of these, like Simulink, use a visual syntax to represent actor-oriented designs. Some are used for designing hardware, some for software, and some for both. An example of a model from Ptolemy II is shown in Fig.2. This model uses the actor defined in Fig.1. Of course, many different syntaxes are compatible with actor-oriented modeling, and for some applications, visual syntaxes like that in Fig.2 are entirely inappropriate.

2.2 Models of Computation

It is important to realize that the syntactic structure of an actor-oriented design says little about the semantics. The semantics is largely orthogonal to

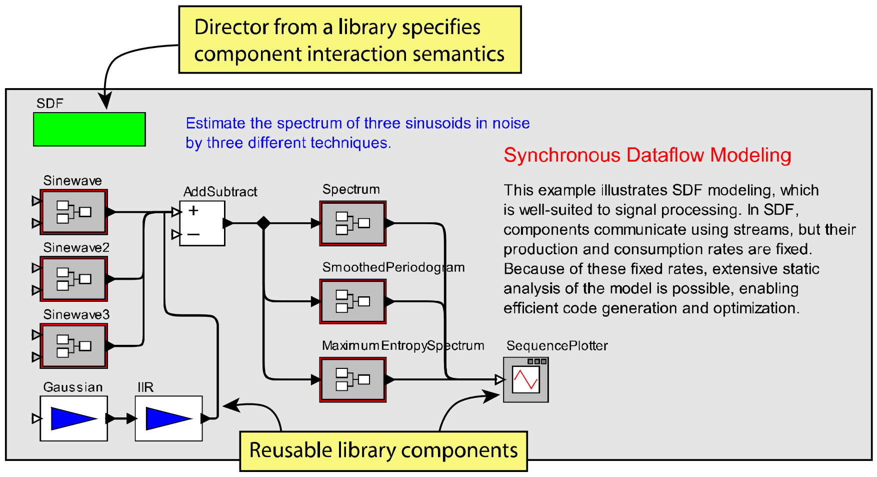
Director from a library specifies component interaction semantics
SDF
Estimate the spectrum of three sinusoids in noise by three different techniques.
Synchronous Dataflow Modeling
This example illustrates SDF modeling, which is well-suited to signal processing. In SDF, components communicate using streams, but their production and consumption rates are fixed. Because of these fixed rates, extensive static analysis of the model is possible, enabling efficient code generation and optimization.
Sinewave
Sinewave2
Sinewave3
Gaussian
IIR
AddSubtract
Spectrum
SmoothedPeriodogram
MaximumEntropySpectrum
SequencePlotter
Reusable library components

Figure 2: Ptolemy II model using the "synchronous dataflow" (SDF) model of computation.

the syntax, and is determined by a *model of computation*. The model of computation defines the nature of communication between components, while the components themselves encapsulate computation. The model of computation consists of operational rules for executing a model. These rules determine when actors perform internal computation, update their internal state, and perform external communication.

There is a wide variety of useful models of computation with different semantics for representing behavior. Additionally, different models of computation can be translated down to lower level platforms in different ways, with different design tradeoffs. A few interesting models of computation are summarized below:

Synchronous Dataflow (SDF) [14]: Actors communicate through queues and send or receive a fixed number of tokens each time they are fired. Actors are fired according to a predetermined static schedule. Synchronous dataflow models are highly analyzable and have been used to describe hardware and software systems.

Synchronous Reactive (SR) [3]: Actors communicate through places (single variables), which may additionally take on the special value *Unknown.* At each execution point, or *tick*, actors update the values of their output, until the system stabilizes. Synchronous reactive models are semantically similar to synchronous languages, which are widely used to describe hardware and software systems.

Continuous Time (CT) [20]: Actors communicate through places, which represent the value of a continuous time signal at a particular point in time. At each time point, actors compute their output based on their previous input and the tentative input at the current time, until the system stabilizes. When combined with actors that perform numerical integration with good convergence behavior, such models are conceptually similar to ordinary differential equations and are often used to model physical processes.

Discrete Event (DE) [13]: Actors communicate through a queue of events in time. Events are processed in global time order, and in response to an event an actor is permitted to emit events at the present or in the future, but not in the past. Discrete event models are widely used to model asynchronous circuits and instantaneous reactions in physical systems.

Giotto [7]: Actors communicate through places and represent independent concurrent processes. The processes are triggered at well-defined periodic points in their execution to update or read the places they are connected to. Giotto models are commonly used to model deterministic execution in a real-time operating system.

In Ptolemy II, the model of computation is indicated by a *director*, represented by the boxes at the upper left of each of Fig.1 and Fig.2. Ptolemy II is unique among the frameworks listed above in that it has no built-in preferred model of computation, but rather supports a variety of models of computation via components called directors. This capability enables hierarchically heterogeneous design [4], where the modeling properties engendered by different models of computation can be combined.

Fig.3 shows a hierarchically heterogeneous Ptolemy II model that uses a continuous time (CT) director at the top level of the hierarchy and a Giotto director at the next level down. The CT director includes an ODE solver and has semantics somewhat similar to Simulink. The Giotto director implements the semantics of the Giotto language [7], which supports periodic hard-real-time tasks and mode switching. The key here is that the Giotto modeling framework (the Giotto director) is not designed specifically to interact with a continuous-time director. Instead, it exports a well-defined behavioral interface for hierarchical composition.

2.3 Relation to other design techniques

In this section, we relate actor-oriented design to other design techniques, using the notion of a *platform*. Sangiovanni-Vincentelli has articulated clearly

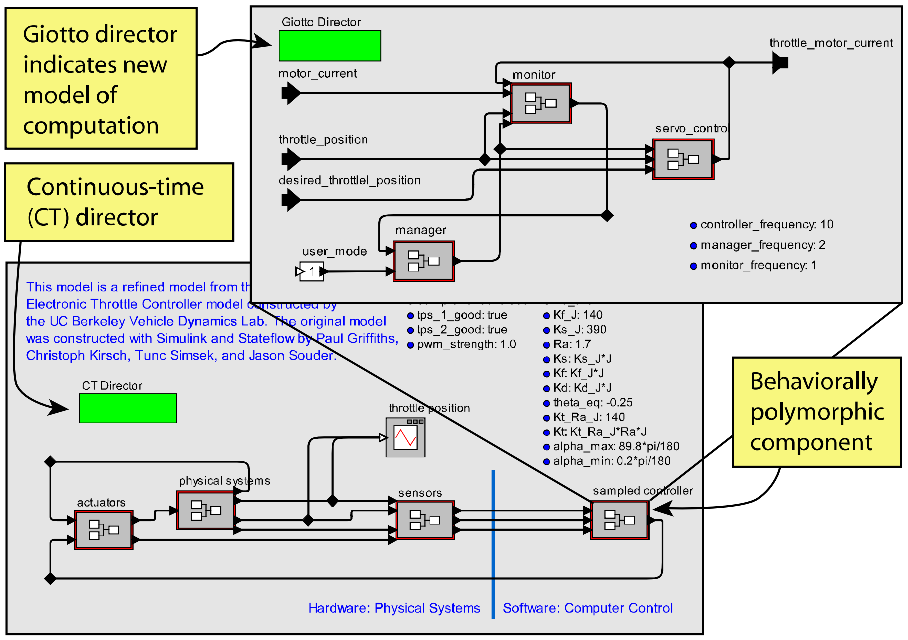
Giotto director indicates new model of computation
Continuous-time (CT) director
Giotto Director
throttle_motor_current
motor_current
monitor
throttle_position
servo_control
desired_throttlel_position
manager
user_mode
1
controller_frequency: 10
manager_frequency: 2
monitor_frequency: 1
This model is a refined model from the Electronic Throttle Controller model constructed by the UC Berkeley Vehicle Dynamics Lab. The original model was constructed with Simulink and Stateflow by Paul Griffiths, Christoph Kirsch, Tunc Simsek, and Jason Souder.
tps_1_good: true
tps_2_good: true
pwm_strength: 1.0
Kf_J: 140
Ks_J: 390
Ra: 1.7
Ks: Ks_J*J
Kf: Kf_J*J
Kd: Kd_J*J
theta_eq: -0.25
Kt_Ra_J: 140
Kt: Kt_Ra_J*Ra*J
alpha_max: 89.8*pi/180
alpha_min: 0.2*pi/180
CT Director
throttle position
physical systems
actuators
sensors
sampled controller
Behaviorally polymorphic component
Hardware: Physical Systems
Software: Computer Control

Figure 3: Hierarchical heterogeneity enables an actor refinement to use a different model of computation than that overall framework.

the benefits of *platform-based design* [25]. We have defined platforms to be sets of designs [15]. Examples of such sets are:

- The set of all boolean functions.
- The set of all x86 binaries.
- The set of syntactically correct Java programs.
- The set of all Java byte-code programs.
- The set of all Wintel PCs.
- The set of all ANSI C programs.

Fig.4 illustrates some platforms and their interrelationships. Each box is a platform. For example, the set of all Java programs has a downward arrow labeled "javac" which represents a function whose domain is the set of all syntactically correct Java programs, and whose range is the set of Java byte

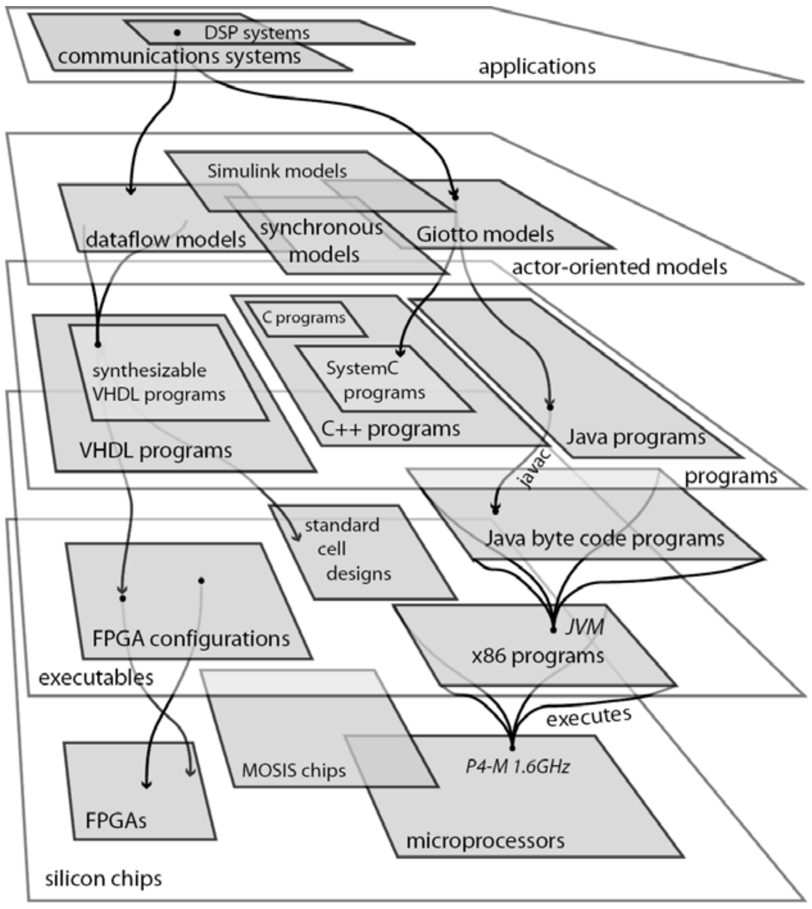

Figure 4: An illustration of some *platforms* (sets of designs) and their interrelationships.

code programs. Similarly, the set of x86 programs contains a program, an implementation the Java virtual machine, that is related to the set of all Java byte code programs by its ability to execute members of the set. For more details about this view of platforms, see [15].

Model-based design [28] concentrates on the specification of designs in platforms with useful modeling properties. For example, Simulink block diagrams represent control systems as visual representations of systems of ordinary differential equations. The set of Simulink designs, therefore, inherits the formal properties and analytical properties of systems of ODEs. These properties are useful to control systems engineers for analyzing, for example, transient responses and stability. Model-based design is also beginning to focus on the modeling of properties themselves via so-called "meta-modeling".

Fig.4 shows a level labeled "actor-oriented models" at a level above programs. The platforms at this level are relatively immature, but play a central role in the future of codesign. In particular, actor-oriented models focus on the separation of concerns between communication and computation [10]. We place actor-oriented models at a level between programs and applications for several reasons. Firstly, actor-oriented design provides a platform which abstracts both hardware and software. Without this level, the gap between "applications" and "programs" is large enough to force very early decisions about hardware vs. software implementations. Secondly, the implementation technologies for early successes in this domain, such as Simulink, take the form of generators (such as Real-Time Workshop) that transform actor-oriented models into program-level models.

2.3.1 Comparison With Object-Oriented Programs

The level in Fig.4 labeled "programs" represents conventional object-oriented software design practice today. Much of the action here is about attempting to give useful modeling properties to designs at this level. For example, UML (the unified modeling language) and the MDA (model-driven architecture) from OMG (the object management group) are about giving modeling structure to designs at this level. Much of the work in design patterns that was kicked off by Gamma *et al*. [5] is about identifying modeling structure in programs at this level.

Fig.5 illustrates the difference between object orientation and actor orientation. In current practice, as defined by languages such as C++ and Java, object-oriented components interact with one another principally by method calls, which represent a transfer of control. The result is that the coordination structure of object-oriented programs is often poorly expressed and described. Managing concurrency, for instance, is notoriously difficult using threads, mutexes and semaphores. Conventions that ensure deadlock avoidance, such as acquisition of locks in a fixed order (see for example [11]), are not supported by the languages (nothing about a method signature declares what locks it will acquire, for instance). As a consequence, these conventions are difficult to apply in practice, and seemingly innocent changes to code can create disastrous failures such as deadlock.

As a result, it is difficult to treat objects in object-oriented languages as *components*. Despite the success of software component technologies such as CORBA and KOALA [21] for representing concurrent software, they fundamentally suffer from fragile composition. The interaction between two components can be broken by simply adding more components to the system. Higher-level patterns are codified only through the APIs of services, and usage patterns for these APIs are expressed only informally in documentation. The communication mechanism for these components is an integral part of a component design, making them difficult to reuse.

In actor-oriented abstractions, low-level implementation mechanisms of threads and semaphores do not even rise to consciousness, forming instead the "assembly-level" mechanisms used to deliver much more sophisticated models of computation. Moreover, actor-oriented abstractions can embrace time and concurrency, and therefore match much better the modeling of hardware systems, which are intrinsically concurrent.

2.3.2 Comparison with Hardware Description Languages

In comparison with object-oriented languages, hardware description languages, such as Verilog and VHDL, are more actor-oriented. A signal in these languages represents the flow of data between concurrent components. For the most part, hardware modules can be composed freely without concern for introducing primitive semantic errors, such as deadlock.

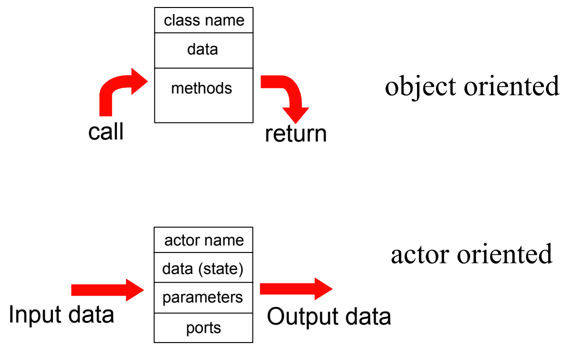

Figure 5: An illustration of the relationship between actor-oriented models and object-oriented components.

However, patterns of communication and interaction in these languages are difficult to specify directly. In order to robustly compose different modules, they must conform to rather arcane interface rules [9]. In other words, the set of signals that a hardware component exposes provides insufficient information for composition.

A key difficulty in hardware description languages arises from the emphasis on clocked synchronous models for synthesis. This emphasis forces designers to describe complex communication behaviors over many clock cycles, complicating module interfaces and the design of modules with truly asynchronous interfaces. Although the description of high-level communication protocols and the use of *communication refinement* can hide some of the complexity, the complexity is still present in the design. In actor-oriented models, high-level models of computation are often sufficient to replace communication protocols over low-level signals. Signals are recovered when the actor oriented model is transformed onto a synchronous platform for synthesis in a process similar to communication refinement. Additionally, the use of high-level models of computation often implies properties of a model that are expensive to check otherwise, such as deadlock-freedom of a set of communication protocols.

A second difficulty in hardware design languages arises from the enforcement of static structure. Although static structure is natural for many hardware applications, such as ASICs, it is inadequate in other cases. For instance, coarse- and fine-grained reconfiguration is often present software and reconfigurable logic designs. Although some kinds of parameterization can be represented manually in hardware design languages, it is left to designers to construct these idioms using registers and control logic. Actor-oriented models not only directly express these kinds of reconfiguration, but also provide for them in a semantically robust way that can be analyzed and constrained when necessary.

2.4 Actors are Components

We consider actors to be *abstract system components* in the following senses:

- They have a well-defined interface, given by ports and parameters, through which they can interact with the rest of the model. Other interactions are not allowed.
- Actor interfaces provide necessary information for checking compatibility with other interfaces, such as data type and behavioral constraints.
- They are generalizable across different uses of the same component. For instance, a component may be given new parameter values or may be connected to a different set of channels.

- They may have vastly different levels of granularity. For instance, an actor may represent algorithms ranging from the boolean OR of two bits to the boolean satisfiability solution of an arbitrary logic formula.
- Larger components may be assembled from smaller components. Component composition is the primary mechanism by which complex behavior is constructed.
- They are executable specifications of behavior. (i.e. Actor models have an operational semantics)

Of these, the generality of actors is something that often varies from one actor to the next. Many actors are designed specifically to fill a particular role in a single model, whereas other actors are designed to be more widely applicable. Although general actors are often more difficult to design, they can be reused in a wider variety of contexts. By maximizing the possibilities for *component reuse*, systems can be designed more quickly and design effort can be leveraged more easily.

Component generality also allows for the possibility of *dynamic reconfiguration* during execution. For example, an actor that is not specialized to a connection with another actor has the potential to be dynamically reconnected to a different actor with no modification. Similarly, an actor that is not specialized to particular parameter values has the potential to be dynamically reconfigured with new values. The use of generalized components increases the available design possibilities when using components.

Object-oriented languages primarily offer generality through variable function arguments. Any set of actual arguments that are compatible with the formal arguments of a function may be passed to that function. Furthermore, a function may be called from many places in a program using different arguments in each case. We call such generality *data polymorphism*. Many object-oriented languages add the notion of *type polymorphism*, where a program can operate uniformly on different types of objects, as long as those types expose similar operations. For the most part, hardware description languages lack rich forms of generality and primarily provide data polymorphism. We identify several types of generality in actor-oriented models, all of which are exhibited in Ptolemy II:

- data polymorphism: An actor may be given data with different values.
- type polymorphism: An actor may be given data of different types.
- parameter generality: An actor may be given a different set of parameter values.
- connection generality: An actor may be connected to a different set of actors.
- behavioral polymorphism: An actor may interact with other actors according to different models of computation.

2.5 Leveraging Static Structure

As mentioned previously, many object-oriented modeling techniques are concerned with expressing the static object-oriented structures. Actor-oriented models also emphasize static structure in a model. We often build models whose structure does not change during the execution of a model, and when we do, we can exploit the fact that the structure is static. Such models have a fixed set of actors, interacting according to a specific model of computation. The ports of an actor form the declared interface of the actor, and this interface does not change. The connections between actors in a model are specified during the construction of the model and do not change during execution of a system. In fact, the Ptolemy II system is specifically designed to allow the construction of models that dynamically evolve during their execution, and yet this capability has been rarely leveraged.

From a software architect's point of view, an emphasis on static structure is a convenient design abstraction. A UML entity-relationship diagram can be considered as the permanent structure of the system, and the fact that this structure is dynamically created in an object-oriented system can often be ignored. Hardware architects, on the other hand, must carefully choose the static computational structure that will be placed in an ASIC since this structure cannot, in most cases, be modified after synthesis. Traditionally, hardware design languages have forced designers to manually provide for circuit reconfiguration.

We have leveraged the static structure of actor-oriented models in two key ways. Firstly, static model structure combined with appropriate type analysis enables detection of unsafe actor compositions. Analysis of both traditional data types and modern behavioral types support generic, reusable components. It is well known how to make such a component polymorphic in a data type sense (*type polymorphism*). It can be designed to be able to add numbers (int, float, double, Complex), add strings (concatenation), add composite types (arrays, records, matrices), and add user-defined types. The formal structure of such type systems is much better known than the formal structure of parameterization and behavioral polymorphism, both of which have re-use benefits at least as great. For this reason, the following sections describe type systems for static analysis of reconfiguration and heterogeneous compositions.

Secondly, we have developed the notion of *actor-oriented specialization* which makes use of the static context of an actor to improve performance in software. Specialization enables practical application of type polymorphism, parameterization, and behavioral polymorphism, since providing these abstractions at run-time incurs significant performance cost. Specialization leverages the static structure in models, as determined through static model analysis, to enable component-based design at all levels of hierarchy. Specialization also enables ASIC and FPGA implementation synthesis without explicit management of reconfiguration.

3 PARAMETERIZATION

Actor parameters are a simple way to configure the behavior of a hierarchical model. At design time, parameters help keep the size of actor libraries manageable and allow models to be quickly modified or tuned for performance. At run time, actor parameters allow for dynamic reconfiguration of actors (and models) while a model is running. Dependencies between parameter values allow simple reconfiguration options to be exposed while hiding the reconfiguration of individual parameters from a designer. However, unconstrained reconfiguration can be problematic during design, since semantic modeling constraints can be violated.

3.1 Representing Reconfiguration

There are many common modeling syntaxes for generically representing reconfiguration in actor-oriented models. For instance, reconfiguration can be represented using a *modal model* controlled by a finite state machine. Each state of the finite state machine corresponds to a hierarchical model that gives the behavior of the modal model in that state. Finite state machine transitions represent switching to a new configuration of a model. Transitions can also set the value of actor parameters. In a hierarchical model, modal models appear as just another level of abstraction. An example of a modal model is shown in Fig.6.

Reconfiguration can also be tied directly to dataflow in a model, using *reconfiguration ports*. A reconfiguration port is associated with an actor parameter and data received by the port reconfigures the value of that parameter. Reconfiguration ports exist in many commercial dataflow environments, such as AVS/Express (Advanced Visual Systems, Inc.). An example of a reconfiguration port is shown in Fig.7.

Reconfiguration can also be represented using a special actor with a single input port. This `SetParameter` actor is bound to a single parameter of a model in a similar fashion to the reconfiguration port. However, since the `SetParameter` actor does not require the presence of hierarchy in the model, it is often more convenient. It can also be more difficult to use, since the point in execution of a model that the `SetParameter` actor is executed is more dependent on its interaction with other actors in the model. One approach to this difficulty is to treat the execution of the `SetParameter` actor as a *request* for reconfiguration at the next possible time, resulting in delayed reconfiguration at a well-defined point in the execution of the model.

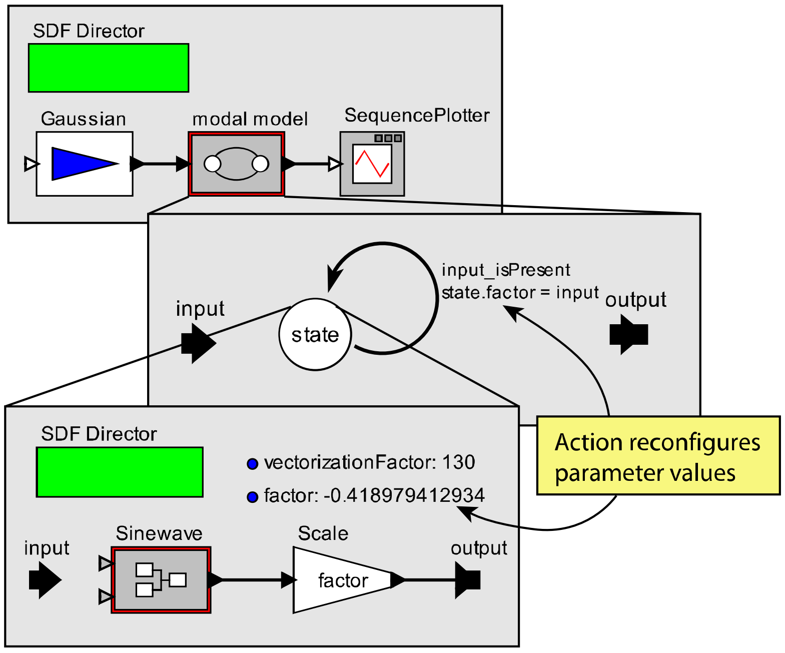

Figure 6: Ptolemy II model using a modal model for reconfiguration.

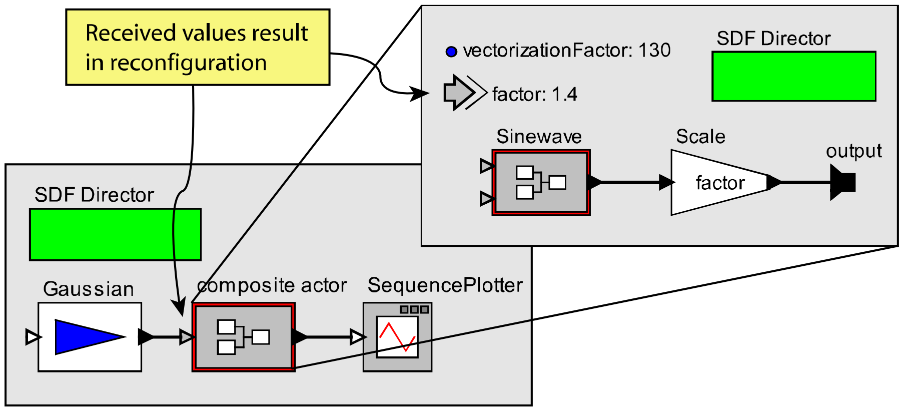

Figure 7: Ptolemy II model illustrating reconfiguration ports.

3.2 Mobile Models

Although most reconfiguration replaces simple data values in a model, we note that reconfiguration in actor-oriented models can also replace portions of an actor-oriented model. The `MobileModel` actor[1], which acts as a placeholder for other components, supports this form of reconfiguration. This actor receives actor-oriented components and behaves as the received component. Fig.8 shows an example of the `MobileModel` actor operating in a model of a control system with a reconfigurable control algorithm [23]. Like more conventional representations of mobile code, as in Java, Ptolemy II allows actors to be transported over the network and safely executed elsewhere. Execution of mobile code relies heavily on strong type systems for ensuring safety. In Java, data type systems are sufficient, while Ptolemy II also relies on behavioral type systems for ensuring safety.

3.3 Constraining Reconfiguration

Generally speaking, reconfiguration modeling syntaxes can modify any parameter in a hierarchical model. However, unconstrained reconfiguration can possibly violate semantic constraints of a model of computation, resulting in unexpected behavior. Unconstrained reconfiguration can also result in unstructured communication outside of a model of computation. For both of these reasons, we would like constrain reconfiguration to ensure understandable models with predictable behavior.

We have focused on *hierarchical reconfiguration* as the key interesting modeling constraint. Informally, hierarchical reconfiguration implies that a

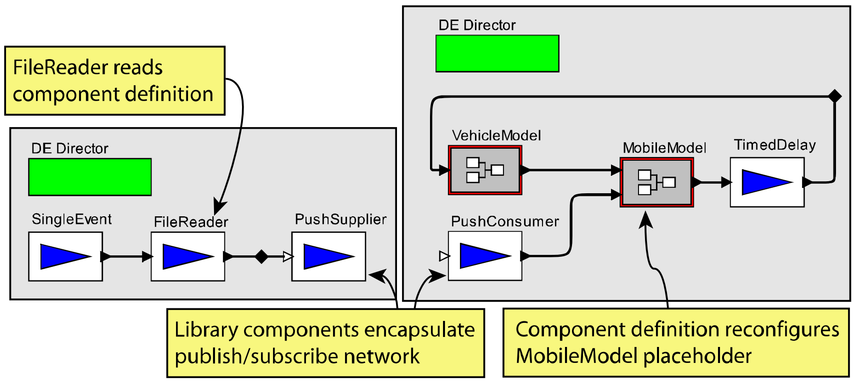
FileReader reads component definition
DE Director
SingleEvent
FileReader
PushSupplier
DE Director
VehicleModel
MobileModel
TimedDelay
PushConsumer
Library components encapsulate publish/subscribe network
Component definition reconfigures MobileModel placeholder

Figure 8: A mobile model is a model that, like mobile code, can be transported over the network and safely executed elsewhere.

1. Designed by Yang Zhao.

parameter associated with a particular actor in a hierarchical model is not reconfigured while that actor is executing. Hierarchical reconfiguration solidifies our intuition that the behavior of an actor is a function of its parameters and the interesting fine-grained interaction of an actor with its environment is strictly expressed through ports.

In order to ensure that reconfiguration in a model is hierarchical, we have developed a *reconfiguration type system* that approximately analyzes how each parameter in a model is reconfigured [24]. This theory determines a *least change context* for each parameter that summarizes the effect of reconfiguration and parameter dependency on that parameter. Reconfiguration constraints, such as hierarchical reconfiguration, can be easily expressed as a constraint on the least change context and efficiently checked. This type system is built-in to Ptolemy II and used to guarantee safe uses of reconfiguration.

4 BEHAVIORAL POLYMORPHISM

The ability to construct hierarchical models using multiple models of computation is a key capability of Ptolemy II, but it creates an interesting challenge. Primarily, a hierarchical component like that in Fig.1 must be able to operate within a foreign model of computation. That is, the semantics of component interaction inside a hierarchical component must be able to differ from the semantics of component interaction outside the hierarchical component. To achieve this, Ptolemy II has pioneered the use of *behavioral polymorphism*. A component is behaviorally polymorphic in that its behavior will depend on the external context in which it is placed. Behavioral polymorphism enables hierarchical heterogeneity.

4.1 Motivating Example

Consider the AddSubtract actor in the center of Fig.2, which adds or subtracts signals that are provided at the input ports. We notice that the function performed by this actor is entirely independent of how data is transported to the actor. For example, the actor can be used in a dataflow framework, where it will add when all connected inputs have data. Or it can be used in a time-triggered framework, where it will add when the clock ticks. Or it can be used in a discrete-event framework, where it will add when any connected input has data, and add in zero time. Or it can be used in a process network framework, where it will execute an infinite loop in a thread that blocks when reading empty inputs and adds when it has read data from all connected inputs. Or it can be used in a CSP-based framework, where it will execute an infinite loop that performs rendezvous on input or output. Etc.

By not choosing among these when defining the component, we get a huge increment in component reusability. Essentially, the interaction pattern between actors is an aspect that is independent from the actors' functionality. Behavioral polymorphism is essential to managing the complexity of hierarchically heterogeneous models. Without behavioral polymorphism, each hierarchical component would have to be designed for a specific combination of the inside and outside models.

A key problem when describing behavioral polymorphism is defining exactly how polymorphic a component actually is. How do we describe the behavioral requirements of a component, in order to ensure that a component will work in a particular circumstance? In a data type system, a type checker ensures that the data type requirements of a component are satisfied in a given context. A corresponding *behavioral type system* is needed to robustly support behavioral polymorphism.

4.2 Object-Oriented Approach to Achieving Behavioral Polymorphism

In Ptolemy II, the director in a model defines the model of computation, which includes the communication mechanism between components. As illustrated in Fig.9, the director instantiates an object that implements a Java interface called Receiver, shown in UML form at the upper right of Fig.9. The six methods of the Receiver interface have different implementations depending on the model of computation. These implementations can, for example,

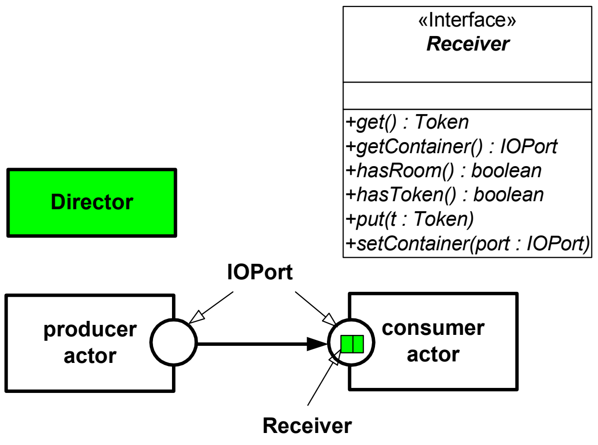

Figure 9: The Ptolemy II Receiver interface.

queue messages, or implement rendezvous, or post events to an event queue, for example. Since the receiver instance used in communication is supplied by the director, not by the component, the component becomes behaviorally polymorphic. Whether a call to the get() method (which reads an input) reads from a queue, blocks, or waits for a rendezvous is not up to the component designer. It is up to the director designer.

4.3 Behavioral Types

The object-oriented approach of the previous section has its limitations. What if:

- The component requires data at all connected input ports?
- The component can only perform meaningful operations on two successive inputs?
- The component can produce meaningful output before the input is known (enabling it to break potential deadlocks)?
- The component has a mutex monitor with another component (e.g. to access a common hardware resource)?

None of these is expressed in the object-oriented interface definition, yet each can interfere with behavioral polymorphism.

The problem is that the Receiver interface only describes object-oriented structure between method calls. It defines abstract data types, but not the dynamic behavior that is central to these communication protocols. Instead of relying on inadequate object-oriented data types, we use a *behavioral type system* (which we previously called a “system-level type system” [16]) to obtain benefits analogous to data typing.

An example of a behavioral type signature for communication protocols is shown in Fig.10. This signature describes a component’s interaction with its environment using an extension of *interface automata* [2][17]. Type checking is done through automata composition, which will detect component incompatibilities. Subtyping order is given by the alternating simulation relation [2], supporting behavioral polymorphism. An alternative representation of behavioral types would be pre/post conditions [19], which may be essentially equivalent, but lacks the intuitive visualization of Fig.10. For details on how to specify behavioral types, see [16][17].

The use of a behavioral type system enables key quality control techniques.

Checking behavioral compatibility of components that are composed. Composition of components that cannot effectively work together because of conflicts in their dynamics (e.g. differing assumptions about communication protocols) is identified as an error. This is analogous to type conflicts in conventional type systems.

Checking behavioral compatibility of components and frameworks. Use of a component in a framework that cannot effectively meet the assumptions of the component is identified as an error.

Behavioral subclassing enables interface/implementation separation. Behavioral type signatures define interfaces that can be implemented by a number of components. For example, components that are simple memoryless stream transformers (which react to input data, transform it in some way, and produce output results) all share the same behavioral interface definition.

Helps with the definition of behaviorally-polymorphic components. Behavioral type signatures declare the minimal assumptions that are required for the component to work. This maximizes the number of contexts in which the component can be used.

5 ACTOR-ORIENTED SPECIALIZATION

Although high-level models can be useful simply for behavioral design and conceptualization, these models become truly powerful when they become a designer's primary programming paradigm. Design tools are more than capable of compiling designs from a modeling platform into a executable platform. Object-oriented modeling tools emphasize round-trip engineering for maintaining consistency between models and code. Many actor-oriented design tools provide code generation capability for automatically translating a model into executable software and hardware. In particular, Simulink's real-

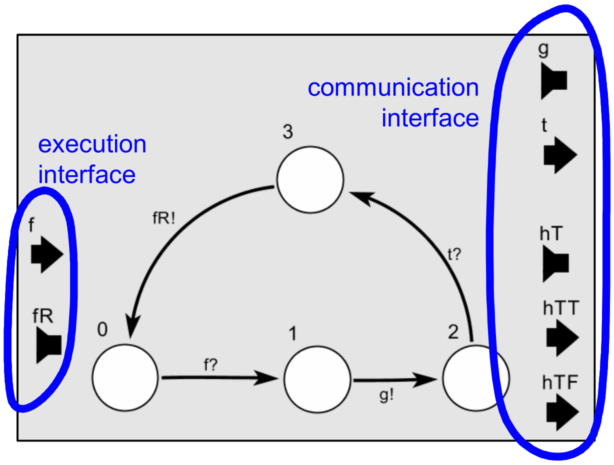

Figure 10: Behavioral type signature using interface automata.

time workshop has become commonly used in the control-systems community.

Specialization of object-oriented programs has grown out of the long history of partial evaluation for functional languages and method inlining optimization in procedural languages. Object-oriented specialization relies on program analysis and extra program annotations to indicate opportunities for specialization [27], and 'optimal' specialization is known to be an undecidable problem. Whereas object-oriented specialization concentrates on transforming the method calls between objects, actor-oriented specialization concentrates on transforming interaction patterns between actors. Because the interaction patterns between actors are explicitly described, actor-oriented specialization is straightforward to apply and relatively simple to understand.

In the context of actor-oriented design, we claim that specialization of a model to a particular context is the primary operation in transforming actor-oriented models into program-level models. Executability alone is not sufficient for resource constrained systems. If it were, then we would simply embed a complete simulation environment in every system and declare success. In reality, optimization to reduce resource usage is crucial for reducing system cost and satisfying timing requirements. Actor-oriented specialization results in optimized behavior while still allowing generic design-time constructs [22]. Specialization implements a transformation from a modeling platform to an efficient implementation platform.

We consider several forms of actor-oriented specialization according to the various generic aspects of Ptolemy II components:

- Parameter specialization
- Type specialization
- Connection specialization
- Domain specialization

Parameter specialization replaces actor parameters that are not reconfigured with a constant value. This specialization relies on analysis of an actor-oriented model to determine constraints on reconfiguration, as described in Section 3. Efficient computation of the values of parameters is possible through partial evaluation of parameter expressions dependent on constant parameters. Additionally, reconfiguration constraints allow for further optimization, since parameter reconfiguration occurs only at well determined points in the execution of a model.

Type specialization replaces generic data types with specific types inferred from the model context. For software implementation, specific types allow the removal of abstract data type indirection through *token unboxing*. In Java implementations synthesized from Ptolemy II, token unboxing greatly reduces load on the garbage collector, resulting in faster and more predictable execution. For hardware implementations, specific types are crucial to selecting efficient representation of data values and operations.

Connection and domain specialization deal directly with the implementation of a model of computation on a particular implementation platform. These specializations are possible when the structure of models and the modeling semantics are fixed at design time. We expect that a provider of a synthesis system will be largely concerned with the efficient implementation of component interaction in a particular system. Given that a designer wishes to target a particular implementation platform, he will select a model of computation that has been provided for that platform and begin designing. Retargeting the model to another implementation platform that provides the same high-level semantics is easy, while targeting an implementation platform without direct support of a model of computation, or migrating a model from one semantics to another is likely to be much more difficult.

6 CONCLUSION

We have outlined a class of design techniques that we call actor-oriented design, and have related it to model-based design, platform-based design, and object-oriented design. We have described mechanisms for generalizing actors, promoting re-use in various contexts. We have focused on parameterization and behavioral polymorphism, and presented described type systems for statically determining valid modeling compositions. These type systems bring to actor-oriented design benefits similar to what abstract data types and their corresponding type systems have brought to object-oriented design. Lastly, we have argued that component specialization can offset the performance costs associated with component-based design.

7 ACKNOWLEDGMENTS

The work described here is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), the Defense Advanced Research Projects Agency (DARPA), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from the State of California MICRO program, and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

References

[1] G. Agha, "Concurrent Object-Oriented Programming," *Communications of the ACM*, 33(9): 125–140, September 1990.

[2] L. de Alfaro and T. A. Henzinger, "Interface Automata," *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering* (FSE), ACM Press, 2001.

[3] Stephen A. Edwards and Edward A. Lee, "The Semantics and Execution of a Synchronous Block-Diagram Language," *Science of Computer Programming*, Vol. 48, no. 1, July 2003.

[4] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, "Taming Heterogeneity---the Ptolemy Approach," *Proceedings of the IEEE*, v.91, No. 2, January 2003.

[5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[6] G. Goessler and A. Sangiovanni-Vincentelli, "Compositional Modeling in Metropolis," Second International Workshop on Embedded Software (EMSOFT), Grenoble, France, LNCS 2491, Springer-Verlag, October 7-9, 2002.

[7] T. A. Henzinger, B. Horowitz and C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming," First International Workshop on Embedded Software (EMSOFT), Lake Tahoe, CA, LNCS 2211, Springer-Verlag, October 8-10, 2001.

[8] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Journal of Artificial Intelligence*, 8(3): 323–363, June 1977.

[9] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-On-A-Chip Designs*, Kluwer, 1998.

[10] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. "System Level Design: Orthogonalization of Concerns and Platform-Based Design", *IEEE Transactions on Computer-Aided Design*, 19(12), December 2000.

[11] D. Lea, Concurrent Programming in Java$^{TM:}$ Design Principles and Patterns, Addison-Wesley, Reading MA, 1997.

[12] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle and P. Volgyesi., "The Generic Modeling Environment," Workshop on Intelligent Signal Processing, May 2001.

[13] E. A. Lee, "Modeling Concurrent Real-time Processes Using Discrete Events," *Annals of Software Engineering, 7:*25–45, 1999.

[14] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, 75(9):55-64, September 1987.

[15] E. A. Lee, S. Neuendorffer and M. J. Wirthlin, "Actor-Oriented Design

of Embedded Hardware and Software Systems," *Journal of Circuits, Systems, and Computers*, 12(3): 231-260, 2003.

[16] E. A. Lee and Y. Xiong, "System-Level Types for Component-Based Design," First International Workshop on Embedded Software (EMSOFT), Lake Tahoe, CA, LNCS 2211, Springer-Verlag, October 8-10, 2001.

[17] E. A. Lee and Y. Xiong, "A Behavioral Type System and Its Application in Ptolemy II," *Formal Aspects of Computing Journal*, special issue on Semantic Foundations of Engineering Design Languages, to appear.

[18] E. A. Lee, "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M03/25*, University of California, Berkeley, July 2, 2003.

[19] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, 16(6): 1811-1841, November 1994.

[20] Jie Liu and Edward A. Lee, "A Component-Based Approach to Modeling and Simulating Mixed-Signal and Hybrid Systems," *ACM Transactions on Modeling and Computer Simulation*, 12(4):343-368, October 2002.

[21] R. van Ommering, "The Koala Component Model for Consumer Electronics Software, *IEEE Computer*, 33(3):78-85, March 2000.

[22] S. Neuendorffer, "Automatic Specialization of Actor-Oriented Models in Ptolemy II," Master's Report, *Technical Memorandum UCB/ERL M02/41*, University of California, Berkeley, CA 94720, December 25, 2002.

[23] S. Neuendorffer, "Implementation Issues in Hybrid Embedded Systems," *Technical Memorandum UCB/ERL M03/22*, University of California, Berkeley, CA 94720, June 2003.

[24] S. Neuendorffer and E. A. Lee, "Hierarchical Reconfiguration of Dataflow Models," Conference on Methods and Models for Codesign (MEMOCODE), June 2004, to appear.

[25] A. Sangiovanni-Vincentelli, "Defining Platform-Based Design," *EEDesign of EETimes*, February 2002.

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

[27] U. P. Schultz, J. L. Lawall and C. Consel, "Automatic Program Specialization for Java," *ACM Transactions on Programming Languages and Systems*, 25(4): 452 - 499, July 2003.

[28] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," IEEE Computer: 110–112, April 1997.