# Model-Driven Development - From Object-Oriented Design to Actor-Oriented Design

Edward A. Lee
UC Berkeley, Berkeley, CA 94720
eal@eecs.berkeley.edu

*Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop) Chicago, Sept. 24, 2003*

***Abstract* -- Most current software engineering is deeply rooted in procedural abstractions. These say little about concurrency, temporal properties, and assumptions and guarantees in the face of dynamic system structure. *Actor-oriented design* contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components. Components called *actors* execute and communicate with other actors. While interfaces in object-oriented design (methods, principally) 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 explores the use of behavioral type systems in actor-oriented design.**

## I. Introduction

Objects in object-oriented design present interfaces consisting principally of methods with type signatures. A method represents a transfer of the locus of control. Much of the talk of "models" in software engineering is about the static structure of object-oriented designs. However, essential properties of real-time systems, embedded systems, and distributed systems-of-systems are poorly defined by such interfaces and by static structure.

### *A. Actor-Oriented Design*

A significantly different direction for embedded software 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* [16]. 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. Primarily, actor-oriented design allows designers to consider the interaction between components distinctly from the specification of component behavior. This contrasts with software component technologies such as CORBA, where interaction between objects is expressed through method invocation. Higher-level patterns are codified only through the API of services, and usage patterns for these APIs are expressed only informally in documentation. As a consequence, the communication mechanism becomes an integral part of a component design.

By focusing on the actor-oriented architecture of systems, we can leverage structure that is poorly described and expressed in procedural abstractions. Managing concurrency, for instance, is notoriously difficult using threads, mutexes and semaphores, and yet even these primitive mechanisms are extensions of procedural abstractions. Conventions that ensure deadlock avoidance, such as acquisition of locks in a fixed order (see for example [7]), 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.

In actor-oriented abstractions, these low-level mechanisms do not even rise to consciousness, forming instead the "assembly-level" mechanisms used to deliver much more sophisticated models of computation.

Our notion of actor-oriented modeling is related to the work of Gul Agha and others. The term "actor" was introduced in the 1970's by Carl Hewitt of MIT to describe the concept of autonomous reasoning agents [6]. 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 are further developing the term to embrace a larger family of models of concurrency that are often more constrained than general message passing. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Moreover, although communication is still through some form of message

passing, it need not be strictly asynchronous.

### B. Platforms

Sangiovanni-Vincentelli has articulated clearly the benefits of *platform-based design* [14]. We have defined platforms to be a set of designs [9]. 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.1 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 code programs. Similarly, the set of x86 programs contains a single 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 [9].

Model-based design [16] is 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.

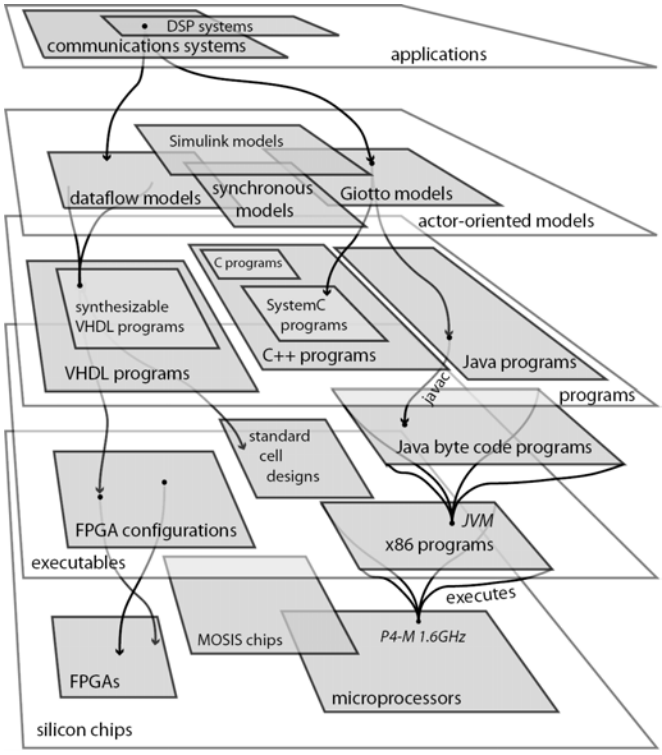


Figure 1: An illustration of some *platforms* (sets of designs) and their interrelationships.

### C. Object-Oriented Programs

The level in Fig.1 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 patters that was kicked off by Gamma *et al*. [3] is about identifying modeling structure in programs at this level.

Actor-oriented design is illustrated in Fig.1 at a level above programs. It is a relatively immature area, with few well-structured programming languages and little software support. But early successes in this domain such as Simulink promise very effective modeling properties combined with effective implementation technologies. The implementation technologies will likely take the form of generators (such as Real-Time Workshop) that transform actor-oriented models into program-level models.

## II. Properties of Actor-Oriented Models

Fig.2 illustrates the difference between object orientation and actor orientation. In current practice, as defined by languages such as C++ and Java, and as represented by abstractions such as UML, object-orientated components interact with one another principally by method calls, which represent a transfer of control. In actor-oriented models, components interact via some sort of messaging scheme that is typically concurrent. The messaging scheme and any constraints on flow of control

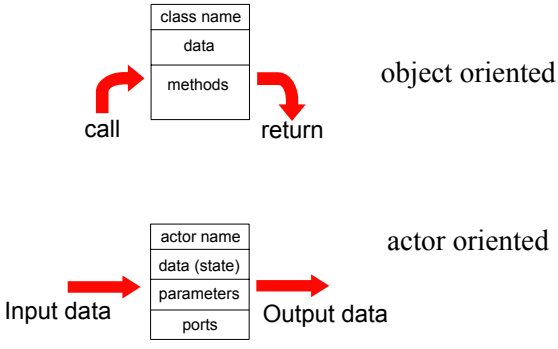


Figure 2: An illustration of the relationship between actor-oriented models and object-oriented components.

together define a *model of computation* that governs the interaction of components.

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. This is *hierarchical abstraction*, illustrated in Fig.3. 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. 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.3, in XML, or in a program designed to a specific

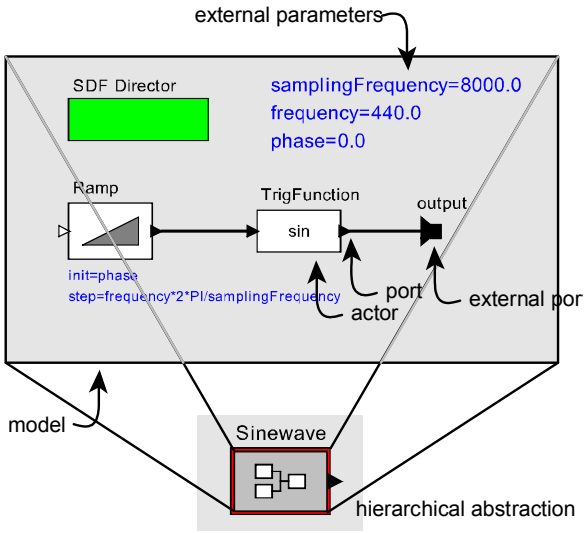


Figure 3: Hierarchical abstraction in actor-oriented design.

API. Ptolemy II [12] offers all three alternatives.

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

There are many examples of actor-oriented languages, frameworks, and software techniques, including Simulink (The MathWorks), Labview (National Instruments), Modelica (Linkoping), GME: actor-oriented meta-modeling (Vanderbilt) [8], Easy5 (Boeing), SPW, signal processing worksystem (Cadence), System studio (Synopsys), ROOM, real-time object-oriented modeling (Rational) [15], VHDL, Verilog, SystemC (Various), Polis & Metropolis (UC Berkeley) [4], and Ptolemy & Ptolemy II (UC Berkeley) [12].

Many of these, like Simulink, use a visual syntax to represent actor-oriented designs. An example of a model from Ptolemy II is shown in Fig.4. This model uses the actor defined in Fig.3. Of course, many different syntaxes are compatible with actor-oriented modeling, and for some applications, visual syntaxes like that in Fig.4 are entirely inappropriate.

In Ptolemy II, the model of computation is indicated by a *director*, represented by the boxes at the upper left of each of Fig.3 and Fig.4. 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 heterogeneous design, where the modeling properties engendered by different models of computation can be combined.

The ability to use multiple models of computation is a key capability of Ptolemy II, but it creates an interesting challenge. First, a hierarchical component like that in Fig.3 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 introduces the notion of *behavioral polymorphism*. A hierarchical component is behaviorally polymorphic in that its behavior will depend on the external context in which it is placed.

## III. Behavioral Polymorphism

### A. Motivating Example

Consider the AddSubtract actor in the center of Fig.4, which adds or subtracts signals that are provided at the input ports. It is well known how to make such a component polymorphic in a data type sense (*data polymorphic*). 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.

Less well known is that it can also be made behaviorally polymorphic. For example, it can be used in a data-flow 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. More importantly, when building hierarchically heterogeneous models, it is essential that hierarchical components be behaviorally polymorphic, or else they would not be truly heterogeneous. Each component would have to be designed for the combination of the inside and outside models.

But how do we ensure that the component will work in all these circumstances? In a data type system, a type checker ensures that a component will work in a given context. A corresponding *behavioral type system* is needed to support behavioral polymorphism.

### B. 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.5, the director instantiates an object that implements a Java interface called Receiver, shown in UML form at the upper right of Fig.5.

The six methods of Receiver shown in Fig.5 have different implementations depending on the model of computation. These implementations can, for example, 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.

### C. Behavioral Types

The object-oriented approach of the previous section

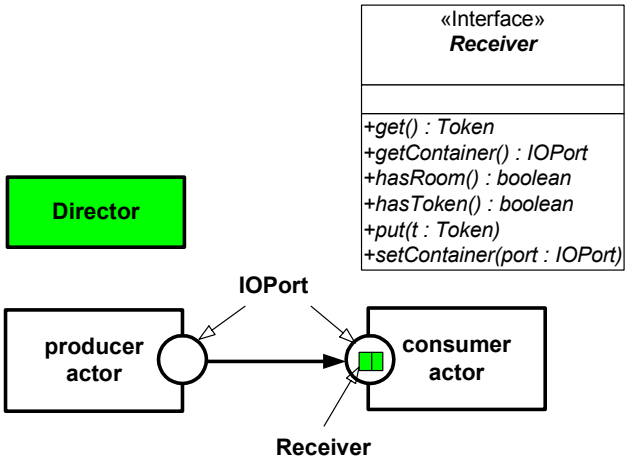


Figure 5: The Receiver interface in Ptolemy II achieves behavioral polymorphism in an object-oriented way.

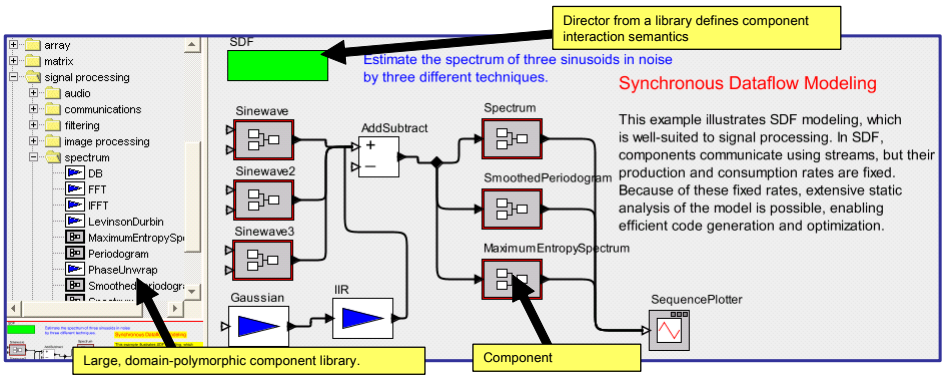


Figure 4: Ptolemy II model using the "synchronous dataflow" (SDF) model of computation.

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 is about static structure. It defines abstract data types, but not dynamic behavior, and yet, communication mechanisms are very much about dynamic behavior. We need to capture the dynamic interaction of components in types. A *behavioral type system* (which we previously called a "system-level type system" [10]) can obtain benefits analogous to data typing.

An example of a behavioral type signature is shown in Fig.6. This captures patterns of component interaction in a type system framework. It describes interaction and component behavior using an extension of *interface automata* [2][11]. 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 [13], which may be essentially equivalent, but lacks the intuitive visualization of Fig.6.

For details on how to specify behavioral types, see [10][11]. In this extended abstract we focus on the consequences of having a behavioral type system.

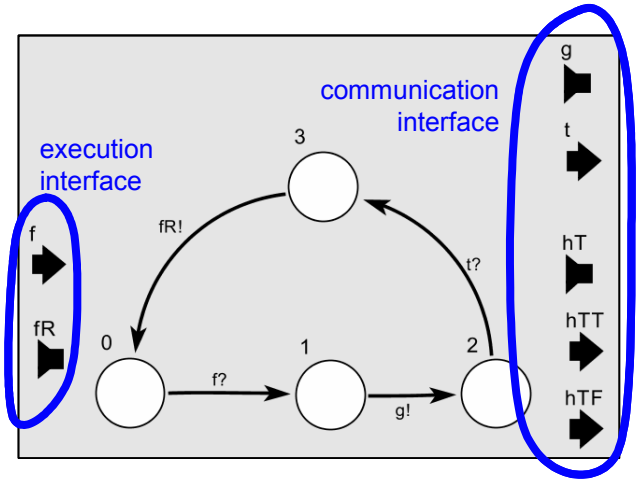


Figure 6: Behavioral type signature using interface automata.

### D. Quality Benefits of using a Behavioral Type System

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

*1) 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.

*2) Checking behavioral compatibility of components and their frameworks.*

Use of a component in a framework that cannot effectively meet the assumptions of the component is identified as an error.

*3) 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.

*4) 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.

### E. Modeling Methods Enabled by Behavioral Type Systems

A number of modeling capabilities are enabled by a behavioral type system.

*1) Hierarchical Heterogeneity*

Fig.7 shows a 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 [5], 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 inter-

act with a continuous-time director. Instead, it is designed to export a behaviorally polymorphic interface when it is composed hierarchically with other models.

*2) Modal Models*

Fig.8 shows a further refinement of the model in Fig.7 where the Giotto model is revealed to have a component that is defined hierarchically as a finite state machine (FSM), where each state of the state machine further refines to another model. This hierarchical composition of FSMs with other models of computation yields a general form of *modal models*, where a mode of operation is represented by a state of the state machine.

Once again, the FSM infrastructure in Ptolemy II is not designed specifically to work with Giotto. Instead, it is designed to export a behaviorally polymorphic interface, and it assumes that the refinements of each of its states (which define the behavior in a mode of operation) are themselves behaviorally polymorphic components.

*3) Mobile Models*

Fig.9 shows a mobile model example[1], which is a model that, like more conventional mobile code, can be transported over the network and safely executed elsewhere. Java, for example, relies heavily on the (data) type system to ensure safety of mobile code. Behavioral type systems provide comparable safety for actor-oriented models.

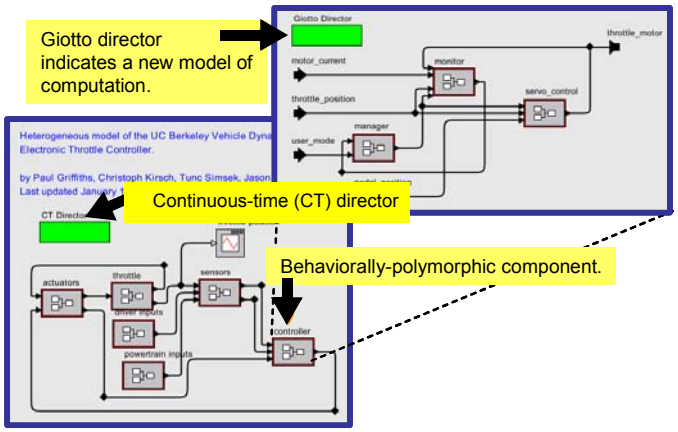


Figure 7: Hierarchical heterogeneity enables an actor refinement to use a different model of computation than that overall framework.

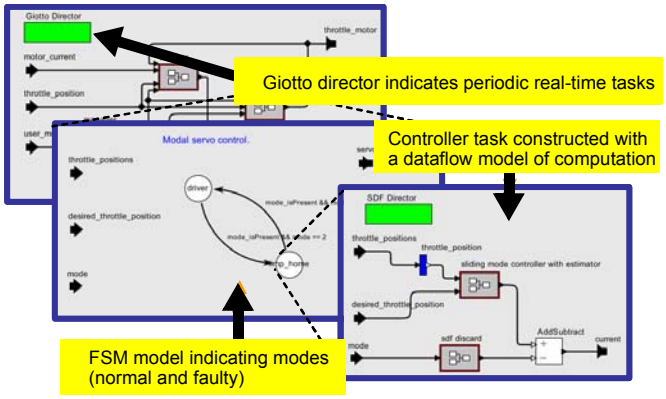


Figure 8: Hierarchical composition of finite state machines (FSMs) with other actor-oriented models yields a general modal model mechanism.

## 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 suggested that a behavioral type system can bring to actor-oriented design benefits similar to what abstract data types and their corresponding type systems have brought to object-oriented design.

## Acknowledgment

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*

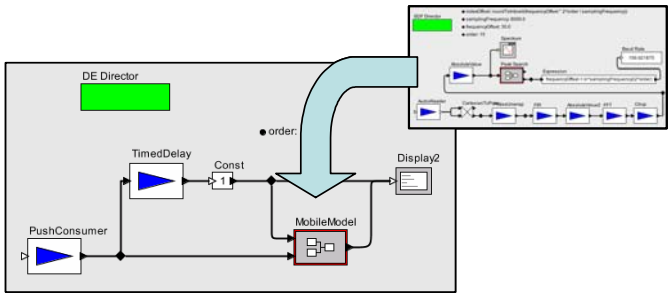


Figure 9: A mobile model is a model that, like mobile code, can be transported over the network and safely executed elsewhere.

[1] Designed by Yang Zhao and Steve Neuendorffer.

*on Foundations of Software Engineering* (FSE), ACM Press, 2001.

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

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

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

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

[7] D. Lea, Concurrent Programming in Java™: Design Principles and Patterns, Addison-Wesley, Reading MA, 1997.

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

[9] 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.

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

[11] 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.

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

[13] 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.

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

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

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