Disciplined Heterogeneous Modeling*
Invited Paper

Edward A. Lee

EECS, UC Berkeley
eal@eecs.berkeley.edu

Abstract. Complex systems demand diversity in the modeling mechanisms. One way to deal with a diversity of requirements is to create flexible modeling frameworks that can be adapted to cover the field of interest. The downside of this approach is a weakening of the semantics of the modeling frameworks that compromises interoperability, understandability, and analyzability of the models. An alternative approach is to embrace heterogeneity and to provide mechanisms for a diversity of models to interact. This paper reviews an approach that achieves such interaction between diverse models using an *abstract semantics*, which is a deliberately incomplete semantics that cannot by itself define a useful modeling framework. It instead focuses on the interactions between diverse models, reducing the nature of those interactions to a minimum that achieves a well-defined composition. An example of such an abstract semantics is the *actor semantics*, which can handle many heterogeneous models that are built today, and some that are not common today. The actor abstract semantics and many concrete semantics have been implemented in Ptolemy II, an open-source software framework distributed under a BSD-style license.

1 Introduction

Occam's razor is a widely used principle in science and engineering that prefers theories and hypotheses with the fewest assumptions, postulates, or entities that are sufficient to accomplish the goal. The principle can be expressed as "entities must not be multiplied beyond necessity" (*entia non sunt multiplicanda praeter necessitatem*) or as "plurality should not be posited without necessity" (*pluralitas non est ponenda sine necessitate*) [1]. The principle is attributed to 14th-century English logician, theologian and Franciscan friar William of Ockham.

* This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part II, LNCS 6395, pp. 273–287, 2010.

Despite its compelling value, the principle has its limitations. Immanuel Kant, for example, felt a need to moderate the effects of Occam's razor, stating "the variety of beings should not rashly be diminished." (*entium varietates non temere esse minuendas*) [2]. Einstein allegedly remarked, "everything should be made as simple as possible, but not simpler" [3].

When applied to design techniques, Occam's razor biases us towards using fewer and simpler design languages and notations. However, experience indicates that redundancy and diversity can both be beneficial. For example, there is benefit to using UML class diagrams even if the information they represent is already encoded in a C++ program, making the diagrams redundant. There is also value in use-case diagrams, which express concepts that are not encoded at all in a C++ program that realizes a design. The use-case diagrams represent aspects of the design not (directly) represented in a class diagram, or even in the program itself.

The plurality of notations in UML and its derivatives runs distinctly counter to the principle in Occam's razor. To take an extreme position, arguably, everything that can be expressed in UML can be expressed in natural language. In fact, historically, natural language specifications of design are the precursors of UML specifications. The syntax of natural languages (at least the Indo-European languages) is simpler and less diverse than the syntax of UML, and it is already familiar to anyone who would engage in design. So what is gained by this anti-razor?

One might postulate that UML diagrams offer more precision than natural language descriptions. This is partly true of some diagrams. Consider class diagrams, for example. These have a rather simple syntax, but are not as precise as a set of C++ header files, for example. In particular, the varied interpretations of associations and the extension mechanisms make it possible to create standard-compliant class diagrams that mean almost anything. In general, UML standards emphasize syntax over semantics, and compliance with the standard is about the structure and appearance of diagrams, and not so much about their meaning. So improved precision can't possibly be the justification for the anti-razor.

Design of software systems is essentially a creative process. Engineers create artifacts that did not previously exist. Occam's razor should not be applied to creative processes. A plurality of media for expression enrich process, even when the plurality is not driven by necessity.

In this paper, I will argue for an anti-razor in a class of design representations called *actor-oriented models*. Actors are concurrent components that share data by sending messages via ports. An amazing variety of techniques have evolved that fit this general description. Were we to apply Occam's razor, we would seek the single unifying actor-oriented modeling language. I argue that instead we need a plurality of distinct actor-oriented modeling languages, together with mechanisms for composing models in these languages, each with a strong and clear semantics. But to support the design of complex systems, we must also provide for heterogenous composition of models.

The technique I describe is that developed in the Ptolemy Project, and largely previously presented in [4,5,6]. The key idea is to use a common abstract syntax for a diverse set of actor-oriented models, and to hierarchically compose those models using an abstract semantics.

This paper is organized as follows. The next section justifies the need for heterogeneous modeling. Section 3 describes the common abstract syntax used in Ptolemy II. Section 4 describes the actor abstract semantics. Section 5 describes a few of the many possible models of computation that have proved useful. Conclusions follow.

2 Heterogeneity

Complex systems demand diversity in the modeling mechanisms. We see this diversity of models very clearly with cyber-physical systems (CPS), which combine computing and networking with physical dynamics, and hence require model combinations that integrate dynamics (often described using differential equations) with models of software. We also see it in applications where timed interactions between components are combined with conventional algorithmic computations, such as in networked computer games. We even see it in traditional software systems when we have concurrent interactions between algorithmic components. One way to deal with a diversity of models is to create very flexible or underspecified modeling frameworks that can be adapted to cover models of interest. The downside of this approach is a weakening of the semantics of the modeling frameworks that compromises interoperability, understandability, and analyzability of the models. An alternative approach is to embrace heterogeneity and to provide mechanisms for a diversity of models to interact.

Model diversity arises in various ways [7]. In *multi-view modeling*, distinct and separate models of the same system are constructed to model different aspects of a system. For example, one model may describe dynamic behavior, while another describes physical design and packaging. In *amorphous heterogeneity*, distinct modeling styles are combined in arbitrary ways within the same model without the benefit of structure. For example, some component interactions in a model may use rendezvous while others use publish-and-subscribe. In *hierarchical multimodeling*, hierarchical compositions of distinct modeling styles are combined to take advantage of the unique capabilities and expressiveness of the distinct modeling styles. A familiar example is Statecharts [8], which hierarchically combines synchronous concurrent composition with finite state machines.

This paper will focus on a disciplined form of hierarchical multimodeling studied in the Ptolemy project and implemented in the Ptolemy II software environment. Using hierarchy, one can effectively divide a complex model into a tree of nested submodels, which are at each level composed to form a network of interacting actors. Our approach constrains each of these networks to be locally

homogeneous, using a common execution and communication semantics called a *model of computation* (MoC). The key is to use constrained MoCs with strong semantic properties as much as possible, and to allow hierarchical composition of distinct MoCs to overcome the constraints that are the price for strong semantic properties.

3 A Common Abstract Syntax

The Ptolemy approach uses a common abstract syntax across diverse models. An abstract syntax defines the structure of models and may be described by a meta model. A meta model for the Ptolemy II abstract syntax is shown in Figure 1. Everything in a Ptolemy II model is an instance of NamedObj, which has a string name (not shown in the meta model). There are four specific kinds of NamedObj: Attribute, Entity, Port, and Relation. Any NamedObj has a name and a collection of instances of Attribute. An Entity contains a collection of instances of Port. Ports are associated with instances of Relation, which mediate connections between ports. A CompositeEntity is an Entity that contains instances of Entity and Relation. An AtomicActor is an executable Entity. A CompositeActor is executable CompositeEntity. A Director is an executable Attribute.

An example is shown in Figure 2 using the concrete syntax of Vergil, a visual editor for Ptolemy II models. In that figure, the top-level of the hierarchy is labeled "Model: CompositeActor," which means that its name is Model and that it is an instance of CompositeActor. Model contains an instance of Director, three actors, and one relation. Actors A and C are composite, whereas actor B is atomic. The ports of the three actors are linked to the relation.

In Figure 2, actor A contains one Port named p, one Director, one actor D, and one Relation. The port of D is linked to the port p via the relation.[1] Actor C is similar except that it does not contain an instance of Director, and it contains an Attribute (indicated with a bullet). In Ptolemy II, some attributes have values given in an expression language. For details, see [9].

A composite actor that contains a director is said to be *opaque*, and one that does not is said to be *transparent*. As I will explain below, opaque composite actors are key to hierarchical heterogeneity. From outside, an opaque composite actor looks just like an atomic actor. Its inside structure cannot be seen. It is a black box. In contrast, a transparent composite actor is simply a syntactic grouping with no semantic meaning. The inside structure is fully visible from outside. It is a white box.

The block diagram in Figure 2 uses one of many possible concrete syntaxes for the same model. The model can also be defined in Java syntax or in an XML schema known as MoML [10]. All three syntaxes describe model structure. We will next give the structure some meaning.

[1] Vergil will typically not show this relation, but it is there in the model structure nonetheless.

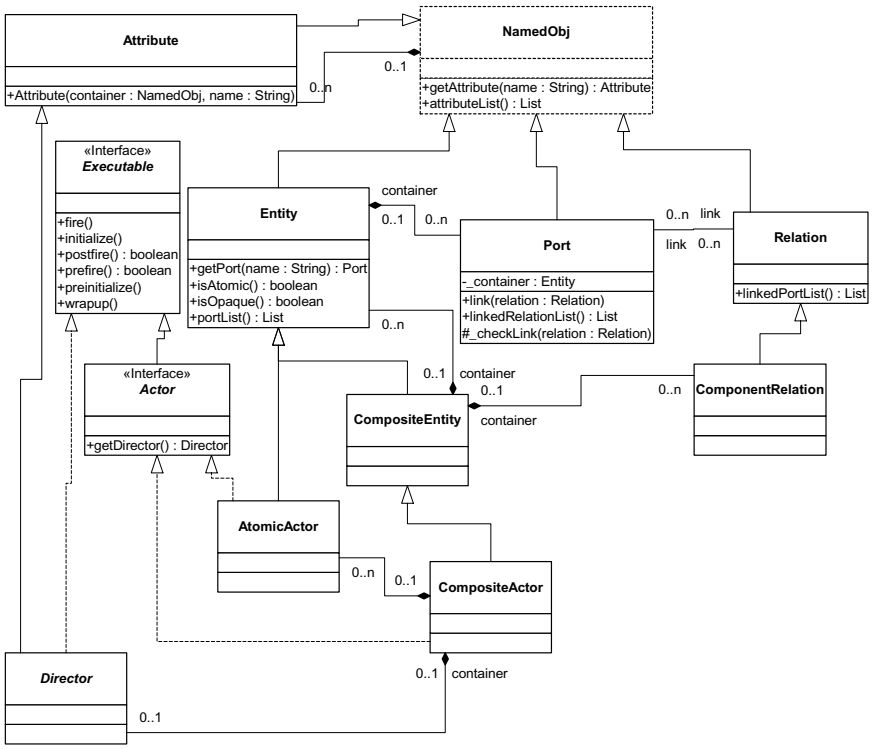

Fig. 1. A meta model for Ptolemy II

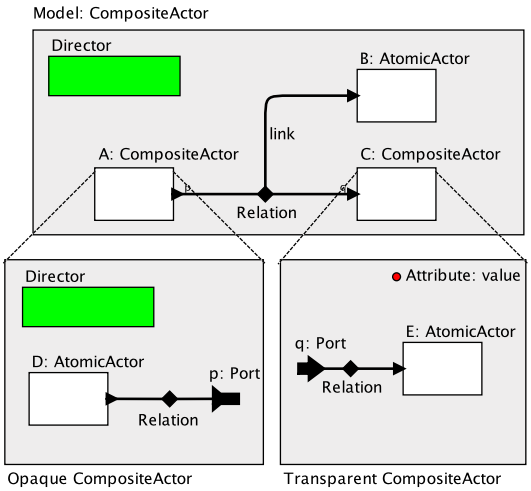

Fig. 2. A hierarchical model in Ptolemy II

4 The Actor Abstract Semantics

In an actor-oriented model, components called *actors* execute concurrently, receiving data from other actors (or from themselves in feedback systems) at their input ports and sending data to other actors (or themselves) via their output ports. What exactly it means to "execute concurrently" and to send or receive data depends on the specific model of computation. The MoC has a *concrete semantics*. To permit hierarchical composition of distinct MoCs, the concrete semantics conforms to an *abstract semantics*. We describe this abstract semantics informally. A formal framework can be found in [11].

The abstract semantics has three distinct aspects, execution control, communication, and models of time. We discuss these in order.

4.1 Execution Control

The semantics of execution and communication is governed by a *director* in a composite actor, which implements a particular MoC. As shown in Figure 1, a Director is an Attribute that implements the Executable interface. It is rendered in Vergil as a green rectangle, as shown in Figure 2. Inserting a Director into a composite actor makes the composite actor executable. Atomic actors also implement the Executable interface.

An execution of an executable actor has the following phases: *setup*, *iterate*, and *wrapup*, where each phase has more fine-grained phases. The setup phase is divided into *preinitialize* and *initialize* phases, methods of the Executable interface. In the preinitialize subphase, an actor performs any actions that may influence static analysis (including scheduling, type inference and checking, code generation, etc.). For example, a composite actor may fix its own internal structure, creating internal actors, etc. The initialization phase initializes parameters, resets local state, and sends initial messages on output ports, if any. Typically, the preinitialization of an actor is performed exactly once during the actor's lifetime in a model execution, and before any other execution of the actor. The initialization of an actor is performed once after the preinitialization and type resolution. It may be performed again if the semantics requires that the actor to be re-initialized, for example, in the hybrid system formalism [12].

Actors perform atomic executions (called *iterations*) in the iterate phase. An iteration is a (typically finite) computation that leads the actor to a quiescent state. The MoC of a composite actor determines how the iteration of one actor is related to the iterations of other actors in the same composite (whether it is concurrent or interleaved, whether and when it is scheduled, etc.).

In order to coordinate the iterations among actors, an iteration is further broken down into three subphases called prefire, fire, and postfire. Prefire tests the preconditions for the actor to execute, such as the presence of sufficient inputs to complete the iteration. The computation of the actor is typically performed during the fire phase, where it will read inputs, process data, and produce outputs. An actor may have persistent state that evolves during execution. The postfire phase is used to update that state in response to any inputs.

Note that despite the fact that computation occurs during the fire phase, the state of the actor is not updated until postfire. This supports fixed-point iteration in some MoCs, such as synchronous reactive models and continuous time differential equations. Directors implementing an MoC with a fixed-point semantics compute the fixed point of actor outputs while keeping the state of each actor constant. The state of an actor can only be updated after the fixed point has been reached. This requires the firing of each actor several times before the actor is postfired. Such a strategy helps to ensure that the MoC is determinate.

This strategy is only followed by actors conforming to the strictest form of the actor abstract semantics. In [13], Goderis et al. classify actor-oriented MoCs into three categories of abstract semantics call *strict*, *loose*, and *loosest*. In the strict actor semantics, prefire, fire, and postfire are all finite computations where only postfire makes any changes to the state of the actor. In the loose actor semantics, changes to the state may be made in the fire phase. In the loosest actor semantics, the fire phase may not be finite. It could represent a non-terminating computation.

An actor that conforms with the strict actor semantics can be used with any actor-oriented director. Such an actor is said to be *domain polymorphic*. An actor that conforms only with the loose actor semantics can be used with fewer directors. An actor that conforms only with the loosest actor semantics can be used with still fewer. Some actors are designed to work only with a single specific type of director. These actors are not domain polymorphic.

A director also implements the same phases of execution. When put within a composite actor, making it opaque, the director endows that composite actor with an executable semantics. If the director conforms to the strict actor semantics, then an opaque composite actor with that director is domain polymorphic. Such directors support the most flexible form of hierarchical heterogeneity in Ptolemy II.

4.2 Communication

A director determines how actors communicate. It does this by creating an object called a *receiver* and placing that object in input ports. There is one receiver for each communication channel. Receivers can implement FIFO queues, mailboxes,

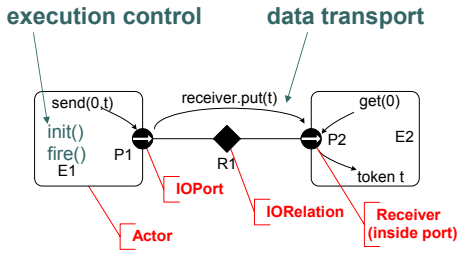

Fig. 3. Communication mechanism in Ptolemy II

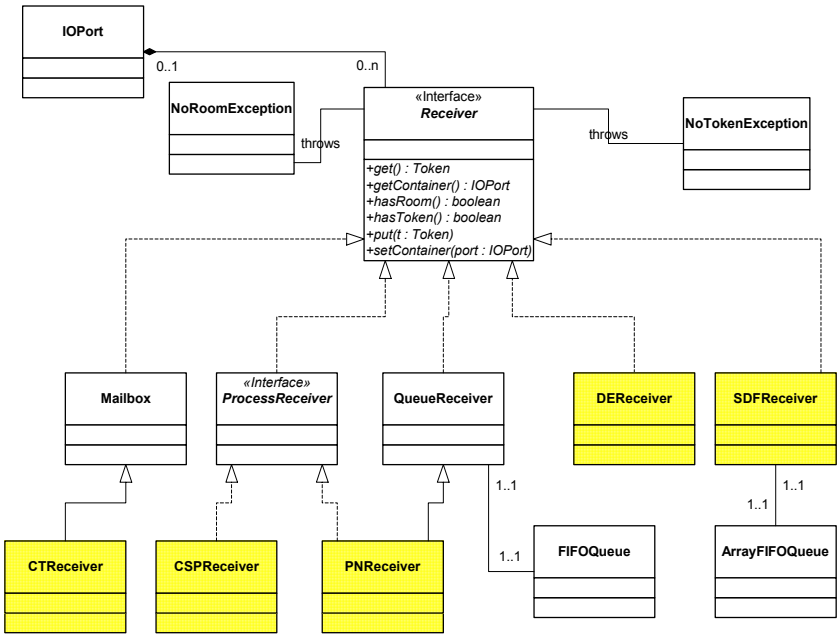

Fig. 4. Meta model for communication in Ptolemy II

proxies for a global queue, rendezvous, etc. They do this by conforming to the meta model shown in Figure 4. When an actor sends data t to an output port p_1, the mechanics of the send are delegated to the receiver in each input port p_2 that the output port is connected to, as shown in Figure 3. When an actor gets data from an input port, that action is also delegated to the receiver. Thus, what it means to send data to an output port of get data from an input port is determined by the director.

4.3 Models of Time

Some models of computation have a notion of time, meaning that they assume that when they are fired, their environment provides a value representing the time of that firing. A part of the actor abstract semantics provides a mechanism for accessing and controlling the advancement of time. Ptolemy II uses a model time known as superdense time [12,14]. Superdense time is represented in Ptolemy II as pair (t, n), where n is of type int and $t = mr$ is an arbitrarily large multiple m of a time resolution r. The multiple m is realized as a Java BigInteger (an arbitrarily large integer), and r is of type double. The pair (t, n) models a physical time t and an *index* n. The index is used to disambiguate simultaneous events that are causally related [12]. Time is ordered lexicographically, which means that $(t_1, n_1) < (t_2, n_2)$ if either $t_1 < t_2$, or $t_1 = t_2$ and $n_1 < n_2$.

When an actor fires, it can ask for the current time. If it does this in the postfire phase of execution, then it is assured that time is nondecreasing. If it does it in the fire phase, there is no such assurance because some directors speculatively advance time while converging to a fixed point [15].

An actor can also request that it be fired at some future time. The director is responsible for ensuring that that the requested future firing occurs. There may be an arbitrary number of intervening firings before the one at that future time.

The model hierarchy is central to the management of time. The top-level director advances time. If the top-level director does not implement a timed model of computation, then by default only the index n is incremented. The current physical time remains at the default start time, with value zero. Timed models of computation may be interleaved in the hierarchy with untimed models of computation (see below). There are certain combinations that do not make sense, however. For example, if the top-level director never advances time, and an actor requests a firing at a future time, then the request cannot be honored. Such combinations result in an exception at run time.

Time can also advance non-uniformly in a model. In particular, modal models support a notion of local time where advancement of time can be temporarily suspended [16]. Within the submodel there is a monotonically non-decreasing gap between local time and environment time. This mechanism is used to model temporary suspension of a submodel.

5 Models of Computation

In Ptolemy II an implementation of a model of computation is called a *domain*.[2] A domain defines the communication semantics and the execution of actors within the domain. We present here some domains that we have realized in Ptolemy II. This is far from a complete list. The intent is to show the diversity of the models of computation under study.

Process network. In the process network (PN) domain, actors represent processes that communicate by (conceptually infinite capacity) FIFO queues [17]. Receivers in this model implement these FIFO queues. Writing to the queues always succeeds, while reading from an empty queue blocks the reader process. The simple blocking-read, nonblocking-write rule ensures the determinacy of the model [18]. This domain is untimed. We have extended the model with specific support for certain forms of nondeterminism. Each actor executes in its own Java thread, so on multi-core machines they can execute in parallel.

Dataflow. Ptolemy II includes several dataflow domains, all restricted special cases of PN [19]. The synchronous dataflow (SDF) domain [20] is a particularly restricted special case with strong, decidable, formal properties. When an actor is executed in this model, it consumes a fixed number of tokens from each input

[2] The term "domain" comes from a fanciful notion in astrophysics, that there are regions of the universe with different sets of laws of physics. A model of computation represents the "laws of physics" of the submodel governed by it.

port, and produces a fixed number of tokens to each output port. A valuable property of SDF models is that deadlock and boundedness can be statically analyzed, and schedules (including parallel schedules) can be statically computed. Receivers in this domain represent FIFO queues with fixed finite capacity, and the execution order of components is statically scheduled. SDF can be timed or untimed. The dynamic dataflow (DDF) domain is more flexible, but computes schedules on the fly.

Discrete event. In the discrete event (DE) domain, actors communicate through events placed on a time line. Each event has a value and a time stamp. Actors process events in chronological order. The output events produced by an actor are required to be no earlier in time than the input events that were consumed. In other words, actors in DE are *causal*.

The execution of this model uses a global event queue. When an actor generates an output event, the event is placed in the queue, and sorted according to its time stamp. Receivers in this domain are proxies for the global event queue. During each iteration of a DE model, the events with the smallest time stamp are removed from the global event queue, and their destination actor is fired. The DE domain supports simultaneous events. At each time where at least one actor fires, the director computes a fixed point [15]. The semantics of DE is given in [21].

Finite-state machines. The finite-state machine (FSM) domain is the only one of the domains discussed here that is not actor oriented [16]. The entities in this domain represent states, and the relations represent transitions between states. Attributes are used to represent guards, which determine when a transition is taken from one state to another. Each state has one port for incoming transitions and one for outgoing transitions. Thus, although FSM models can be represented with the same abstract syntax as actor models, the ports are not used for communication but rather for sequential control.

The FSM domain interoperates with all other domains hierarchically, providing a powerful construct called a *modal model* [16]. In a modal model, each state of an FSM represents a mode of execution. The state can have one or more *refinements*, which are submodels with a director that are active when the FSM is in that state. When a submodel is not active, its local time does not advance.

Continuous time. The *Continuous* domain [22,12] models ordinary differential equations (ODEs), extended to allow the handling of discrete events. Special actors that represent *integrators* are connected in feedback loops in order to represent the ODEs. Each connection in this domain represents a continuous-time function, and the components denote the relations between these functions.

Each receiver in the Continuous domain is a buffer with size one. It contains the value of the continuous function of the corresponding connection at a specific time instant. The execution of a Continuous model involves the computation of a numerical solution to the ODEs. In an iteration of a Continuous model, time is advanced by a certain amount determined by the solver. At each instant, the director computes a fixed point for all signal values using the principles

of synchronous/reactive models (see below) [15]. To advance time, the director chooses a time step with the help of a solver and speculatively executes actors through this time step. If the time step is sufficiently small (key events such as level crossings, mode changes, or requested firing times are not skipped over), then the director commits the time increment and proceeds to the postfire phase.

The Continuous director conforms to the strict actor semantics, and hence interoperates with all other timed Ptolemy II domains. Combining it with FSMs yields a particular form of modal model known as a *hubrid system* [12,16]. Combinations with discrete-event and synchronous/reactive are also particularly useful [15].

Redezvous. In the Rendezvous domain, actors represent processes that communicate by atomic instantaneous rendezvous. Receivers in this domain implement the rendezvous points. An attempt to put a token into a receiver will not complete until a corresponding attempt is made to get a token from the same receiver, and vice versa. As a consequence, the process that first reaches a rendezvous point will stall until the other process also reaches the same rendezvous point [23]. Like PN, this domain supports explicit nondeterminism.

6 Related Work

The actor-oriented models I consider here are syntactically related to composite structure diagrams of UML 2 [24,25], or more directly its derivative SysML [26]. The internal block diagram notation of SysML, which is based on the UML 2 composite structure diagrams, particularly with the use of flow ports, is closely related to actor models. In SysML, the actors are called "blocks," presumably because the term "actor" was already in use in UML for another purpose. SysML, however, defines the syntax of these diagrams, not their semantics. Although the intention is that "flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions" [26], there is nothing like an MoC given in SysML. Therefore, the same SysML diagrams may in fact represent many different designs. Standardizing the notation is not sufficient to achieve effective communication among designers. MARTE (modeling and analysis of real-time and embedded systems) goes a bit further [27], but specifically avoids "constraining" (or even defining) the execution semantics of models. This flexibility may account for some of the success of these notations, but it is arguable that constraints that lead to well-defined and interoperable models have value as well.

Our notion of actor-oriented modeling is related to the term *actor* as introduced in the 1970's by Hewitt to describe the concept of autonomous reasoning agents [28]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [29]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. The Ptolemy version embraces 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.

Some authors use the term multi-paradigm modeling for the mixing of models of computation [30]. In this paper, we focus on techniques that combine actors with multi-paradigm modeling. An early systematic approach to such mixed models was realized in Ptolemy Classic [4]. Ptolemy II is a successor to Ptolemy Classic [5]. Influenced by the Ptolemy approach, SystemC is capable of realizing multiple MoCs [31,32]. So are ModHel'X [33] and ForSyDe [34,35].

Another approach supports mixing concurrency and communication mechanisms without the structural constraints of hierarchy [36,37]. A number of other researchers have tackled the problem of heterogeneity in creative ways [38,39].

For each MoC, a Ptolemy model contains a director, which annotates the model to assert the MoC being used, and provides either a code generator or an interpreter for the MoC (or both). An interesting alternative is given by "42" [40], which integrates with the model a specification of a customized MoC.

Mathematical foundations of heterogeneous actor models are given by the tagged-signal model [11]. The model offers a declarative semantics that facilitates comparing variants and defining heterogeneous composition.

Several of the above mentioned tools and techniques are extensible in that there is no fixed finite set of MoCs. Several widely used tools, in contrast, provide fixed combinations of a few MoCs. Commercial tools include Simulink/StateFlow (from The MathWorks), which combines continuous and discrete-time actor models with finite-state machines, and LabVIEW (from National Instruments), which combines dataflow actor models with finite-state machines and with a time-driven MoC. Statemate [41] and SCADE [42] combine finite-state machines with a synchronous/reactive formalism [43]. Giotto [44] and TDL [45] combine FSMs with a time-driven MoC. Several hybrid systems modeling and simulation tools combine continuous-time dynamical systems with FSMs [46].

The Y-chart approach approach supports heterogeneous modeling and is popular for hardware-software codesign [47]. This approach separates modeling of the hardware architecture that will realize a system from modeling of application behavior (a form of multi-view modeling), and provides mechanisms for bringing these disparate models together. These mechanisms support explorations of the design space that can trade off hardware cost and complexity with software design. Metropolis is a particularly elegant tool realizing this approach [36]. It introduces a notion called a "quantity manager" that mediates interactions between functionality and the resources required to implement the functionality.

Modelica [48] also has actor-like semantics in that components are concurrent and communicate via ports, but the ports are neither inputs nor outputs. Instead, the connections between ports declare equation constraints on variables. This approach has significant advantages, particularly for specifying physical models based on differential-algebraic equations (DAEs). However, the approach also appears to be harder to combine heterogeneously with other MoCs.

7 Conclusion

This paper reviews actor-oriented modeling of complex systems, arguing that it provides a disciplined approach to heterogeneity. The central notion in hierarchical model decomposition is that of a *domain*, which implements a particular model of computation. Technically, a domain serves to separate the flow of control and data between components from the actual functionality of individual components. Besides facilitating hierarchical models, this factoring potentially also dramatically increases the reusability of components and models.

Acknowledgements

Hundreds of people have contributed to Ptolemy II and its predecessor, Ptolemy Classic. For the particular topics in this paper, I give credit to Christopher Brooks, Joe Buck, Thomas Huining Feng, Soonhoi Ha, Jie Liu, Xiaojun Liu, Dave Messerschmitt, Stephen Neuendorffer, Tom Parks, John Reekie, Yuhong Xiong, and Haiyang Zheng.

References

1. Encyclopedia Britannica: Ockham's razor. Encyclopedia Britannica (retrieved June 24, 2010)
2. Smith, N.K.: Immanuel Kant's Critique of Pure Reason. Macmillan and Co., Basingstoke (1929)
3. Shapiro, F.R.: The Yale Book of Quotations. Yale University Press, New Haven (2006)
4. Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogeneous systems. Int. Journal of Computer Simulation, special issue on "Simulation Software Development" 4, 155–182 (1994)
5. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE 91(2), 127–144 (2003)
6. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers 12(3), 231–260 (2003)
7. Brooks, C., Cheng, C., Feng, T.H., Lee, E.A., von Hanxleden, R.: Model engineering using multimodeling. In: International Workshop on Model Co-Evolution and Consistency Management (MCCM), Toulouse, France (2008)
8. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)
9. Brooks, C., Lee, E.A., Neuendorffer, S., Zhao, Y., Zheng, H.: Heterogeneous concurrent modeling and design in Java. Technical Report Technical Memorandum UCB/EECS-2008-28, University of California (April 1, 2008)
10. Lee, E.A., Neuendorffer, S.: MoML - a modeling markup language in XML. Technical Report UCB/ERL M00/12, UC Berkeley (March 14, 2000)
11. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Circuits and Systems 17(12), 1217–1229 (1998)

12. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 25–53. Springer, Heidelberg (2005)
13. Goderis, A., Brooks, C., Altintas, I., Lee, E.A., Goble, C.: Heterogeneous composition of models of computation. Future Generation Computer Systems 25(5), 552–560 (2009)
14. Liu, X., Matsikoudis, E., Lee, E.A.: Modeling timed concurrent systems. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 1–15. Springer, Heidelberg (2006)
15. Lee, E.A., Zheng, H.: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: EMSOFT, Salzburg, Austria. ACM, New York (2007)
16. Lee, E.A.: Finite state machines and modal models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley (November 1, 2009)
17. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE 83(5), 773–801 (1995)
18. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: Gilchrist, B. (ed.) Information Processing, pp. 993–998. North-Holland Publishing Co., Amsterdam (1977)
19. Lee, E.A., Matsikoudis, E.: The semantics of dataflow with firing. In: Huet, G., Plotkin, G., Lévy, J.J., Bertot, Y. (eds.) From Semantics to Computer Science: Essays in memory of Gilles Kahn. Cambridge University Press, Cambridge (2009)
20. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
21. Lee, E.A.: Modeling concurrent real-time processes using discrete events. Annals of Software Engineering 7, 25–45 (1999)
22. Liu, J.: Continuous time and mixed-signal simulation in Ptolemy II. Memo M98/74, UCB/ERL, EECS UC Berkeley, CA 94720 (July 1998)
23. Hoare, C.A.R.: A theory of CSP. Communications of the ACM 21(8) (August 1978)
24. Bock, C.: SysML and UML 2 support for activity modeling. Systems Engineering 9(2), 160–185 (2006)
25. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1998)
26. Object Management Group (OMG): System modeling language specification v1.1. Technical report, OMG (2008)
27. (OMG), O.M.G.: A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG (August 2007)
28. Hewitt, C.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence 8(3), 323–363 (1977)
29. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7(1), 1–72 (1997)
30. Mosterman, P.J., Vangheluwe, H.: Computer automated multi-paradigm modeling: An introduction. Simulation: Transactions of the Society for Modeling and Simulation International Journal of High Performance Computing Applications 80(9), 433–450 (2004)
31. Patel, H.D., Shukla, S.K.: SystemC Kernel Extensions for Heterogeneous System Modelling. Kluwer, Dordrecht (2004)
32. Herrera, F., Villar, E.: A framework for embedded system specification under different models of computation in SystemC. In: Design Automation Conference (DAC), San Francisco. ACM, New York (2006)

33. Hardebolle, C., Boulanger, F.: ModHel'X: A component-oriented approach to multi- formalism modeling (October 2, 2007)
34. Jantsch, A.: Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation. Morgan Kaufmann, San Francisco (2003)
35. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Circuits and Systems 23(1), 17–32 (2004)
36. Goessler, G., Sangiovanni-Vincentelli, A.: Compositional modeling in Metropolis. In: Second International Workshop on Embedded Software (EMSOFT), Grenoble, France. Springer, Heidelberg (2002)
37. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: International Conference on Software Engineering and Formal Methods (SEFM), Pune, pp. 3–12 (2006)
38. Burch, J.R., Passerone, R., Sangiovanni-Vincentelli, A.L.: Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In: International Conference on Application of Concurrency to System Design, p. 13 (2001)
39. Feredj, M., Boulanger, F., Mbobi, A.M.: A model of domain-polymorph component for heterogeneous system design. The Journal of Systems and Software 82, 112–120 (2009)
40. Maraninchi, F., Bhouhadiba, T.: 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In: 6th ACM International Conference on Generative Programming and Component Engineering (GPCE), Salzburg, Austria, pp. 1–3 (2007)
41. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: STATEMATE: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering 16(4) (1990)
42. Berry, G.: The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies (2003)
43. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. Proceedings of the IEEE 79(9), 1270–1282 (1991)
44. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, p. 166. Springer, Heidelberg (2001)
45. Pree, W., Templ, J.: Modeling with the timing definition language (TDL). In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) ASWSD 2006. LNCS, vol. 4922, pp. 133–144. Springer, Heidelberg (2008)
46. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and tools for hybrid systems design. Foundations and Trends in Electronic Design Automation 1(1/2) (2006)
47. Kienhuis, B., Deprettere, E., van der Wolf, P., Vissers, K.: A methodology to design programmable embedded systems. In: Deprettere, F., Teich, J., Vassiliadis, S. (eds.) SAMOS 2001. LNCS, vol. 2268, p. 18. Springer, Heidelberg (2002)
48. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley, Chichester (2003)