Classes and Inheritance in Actor-Oriented Design

Edward A. Lee
Xiaojun Liu
Stephen Andrew Neuendorffer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-154
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-154.html

November 21, 2006

Acknowledgement

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 award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota.

Classes and Inheritance in Actor-Oriented Design

EDWARD A. LEE
UC Berkeley
and
XIAOJUN LIU
Sun Microsystems
and
STEPHEN NEUENDORFFER
Xilinx

Actor-oriented components emphasize concurrency and temporal semantics and are used for modeling and designing embedded software and hardware. Actors interact with one another through ports via a messaging schema that can follow any of several concurrent semantics. Domain-specific actor-oriented languages and frameworks are common (Simulink, LabVIEW, SystemC, etc.). However, they lack many modularity and abstraction mechanisms that programmers have become accustomed to in object-oriented components, such as classes, inheritance, interfaces, and polymorphism. This paper shows a form that such mechanisms can take in actor-oriented components, gives a formal structure, and describes a prototype implementation.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms: Embedded Systems, Actor-Oriented Design

Additional Key Words and Phrases: actors, components, interfaces, type systems

1. INTRODUCTION

Actor-oriented design is a component methodology that has proven effective for embedded software, hardware design, and domain-specific modeling. Components that we call *actors* execute and communicate with other actors in a *model*, as illustrated in figure 1. Actors have a well defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes *ports* that represent points of communication for an actor, and *parameters* which are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed. 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, a *model* may also define

This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota.

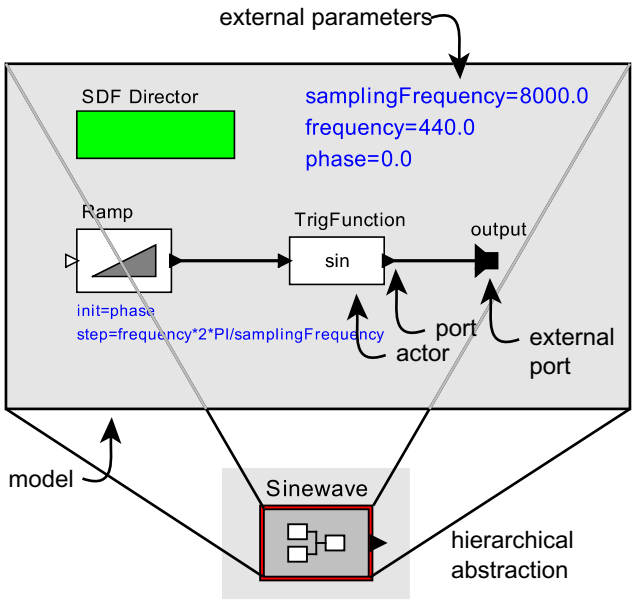

Fig. 1. Illustration of an actor-oriented model (above) and its hierarchical abstraction (below).

an external interface, which we call its *hierarchical abstraction*. This interface consists of *external ports* and *external parameters*, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that compose the model. External parameters of a model can be used to determine the values of the parameters of actors inside the model. A model, therefore, is an actor.

Taken together, the concepts of models, actors, ports, parameters and channels describe the *abstract syntax* of actor-oriented design (this has also been called "static semantics" [Ledeczi et al. 2001] and "structural semantics" [Jackson and Sztipanovits 2006]). This syntax can be represented concretely in several ways, such as graphically, as in figure 1, in XML [Lee and Neuendorffer 2000], in a program designed to a specific object-oriented API, as in SystemC, which has roots in Scenic [Liao et al. 1997], or in an actor-oriented language, such as StreamIT [Thies et al. 2002] or Cal [Eker and Janneck 2003]. Ptolemy II [Eker et al. 2003] offers all four alternatives.

Actor-oriented languages and frameworks are growing in popularity, particularly for embedded software, hardware, and domain-specific design. Actor-oriented design easily falls within early conceptions of what object-oriented design is about. But in the last two decades or so, OO design has crystalized into a style, represented by Java, C#, C++, and UML, that distinctly lacks the features we need, such as managing concurrency [Johnson 1994]. So-called "active objects" are a step in the right direction, but they are little more than disguised threads, and offer little help to the designer. Threads are a notoriously poor concurrency model for reliable systems [Lee 2006]. Our objectives in this paper are to offer a component technology

that brings all the benefits of OO design but with much better concurrency models.

Actor-oriented design has been around since at least 1966, when Bert Sutherland used one of the first acknowledged object-oriented frameworks, Sketchpad [Sutherland 1963], created by his brother Ivan Sutherland, to build the first actor-oriented programming language (which had a visual syntax) [Sutherland 1966]. Today, actor-oriented languages and frameworks often have visual syntaxes (e.g. Simulink and LabVIEW), and are frequently built on top of object-oriented languages in order to leverage their modularity mechanisms [Buck et al. 1994].

AO languages, like OO languages, are about modularity of software. In AO design, components are concurrent objects that communicate via messaging, as opposed to abstract data structures that interact via procedure calls. Although AO languages frequently inherit the OO modularity mechanisms of the languages on which they are built [Buck et al. 1994], these mechanisms have largely not been adapted to operate at the level of AO design. We will show that many (if not all) of the innovations of OO design, including concepts such as the separation of interface from implementation, strong typing of interfaces, subtyping, classes, and inheritance, be adapted to operate at the level of AO design. We describe an implementation of these mechanisms in Ptolemy II, illustrate the mechanisms with a simple example, and outline their formal structure.

1.1 Models of Computation

It is important to realize that the syntactic structure of an actor-oriented language says little about the semantics. The semantics is largely orthogonal to the syntax, and is determined by a *model of computation* [Lee et al. 2003]. 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.

Examples of models of computation that have been used in AO languages include the continuous-time semantics of Simulink (from The MathWorks), the dataflow semantics of LabVIEW (from National Instruments), and the discrete-event semantics of OPNET Modeler (from OPNET Technologies). These models of computation form the *design patterns of component interaction*, in the same sense that Gamma, *et al.* describe design patterns in OO languages [Gamma et al. 1994]. Many such systems have visual syntaxes and are often viewed more as modeling tools than as programming languages; in this paper, we consider these software systems to be editors, interpreters, and compilers for actor-oriented programming languages, and indeed they are increasingly often used in this way, to develop embedded software for example.

The techniques described in this paper apply broadly to AO design, independent of the model of computation. We have tested them in the Ptolemy II framework with continuous-time, discrete-event, dataflow, and process network semantics, and several more experimental models of computation. They work in all of these because they operate at the level of the abstract syntax, not at the level of the concurrent semantics. Moreover, even without any emphasis on efficiency or scalability, the prototype has proven capable of handling thousands of components efficiently.

2. RELATED WORK

2.1 Software Components

Prevailing software component architectures such as CORBA, dot Net, and J2EE, are deeply rooted in the procedural semantics of the dominant object-oriented languages C++, C#, and Java. In such procedural semantics, concurrency is managed using threads, monitors and semaphores, a notoriously difficult approach [Lee 2006]. As a result, in concurrent systems, it is difficult to treat objects in object-oriented languages as components since they suffer from fragile composition. The interaction between two components can be broken by simply adding more components to the system. Higher-level patterns, such as the CORBA event service, are codified only through object-oriented API, and usage patterns for these APIs are expressed only informally in documentation. The communication mechanism for components becomes 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. The functionality of components is separated from their communication mechanisms, as is advocated by many researchers [Keutzer et al. 2000; Gssler and Sifakis 2005]. Moreover, actor-oriented abstractions can embrace time and concurrency, and therefore match much better the modeling of embedded systems, which are intrinsically concurrent.

2.2 Actor-Oriented Design

Our notion of AO 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 [Hewitt 1977]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [Agha 1986; 1990; Agha et al. 1993; Agha et al. 1997]. 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. The term "actor" has also been used since the mid 1970s to describe components in dataflow models of computation [Dennis 1974].

A number of more recent efforts adopt an actor-oriented approach. ROOM (Real-time Object-Oriented Modeling [Selic et al. 1994]) from Rational Software (now IBM) extends OO components with ports and concurrent semantics and has influenced the development of "Capsules" in UML-RT and "Composite Structures" in UML-2.* Port-based objects [Stewart et al. 1997], I/O automata [Lynch 1996] and hybrid I/O automata [Lynch et al. 1996], Moses [Esser and Janneck 2001], Metropolis [Goessler and Sangiovanni-Vincentelli 2002], Ptolemy [Buck et al. 1994]

*UML had already claimed the term "actors" in use-case diagrams, and hence could not use the term for these concurrent objects.

and Ptolemy II [Eker et al. 2003] all emphasize actor orientation. Languages for designing actors are a current research topic; for example StreamIT [Thies et al. 2002], which calls actors "filters," and Cal [Eker and Janneck 2003] are languages for designing hardware and software components that interact with dataflow semantics.

2.3 Prototypes and Classes in Actor-Oriented Languages

This paper is about extending actor-oriented design techniques with modularity mechanisms like those in OO languages. A number of interesting experiments in this direction have been performed by others. The GME system from Vanderbilt [Karsai 1995] has been extended to support actor-oriented prototypes [Karsai et al. 2003]. This work is the closest that we have found to what is described in this paper, and we will have more to say about it below.

Some older projects also extend actor-oriented models with modularity methods. CodeSign [Esser 1996], from ETH builds in an OO notion of classes into a design environment based on time Petri nets. Concurrent ML [Reppy 1991], with its synchronous message passing between threads built in a functional style with continuations, can also be viewed as an actor-oriented framework, and has well-developed modularity mechanisms. In real-time object-oriented modeling (ROOM) [Selic et al. 1994], ports have protocol roles that are abstract classes defining behavior that the port implements. Each of these mechanisms, however, is tightly bound to a particular concurrent semantics. This paper is about defining modularity mechanisms for a broad spectrum of actor-oriented semantics. It accomplishes this by defining these mechanisms at the level of the abstract syntax. Our hope is that the next generation of domain-specific frameworks beyond Simulink and LabVIEW will inherit these modularity mechanisms, and that because these mechanisms are independent of the concurrent semantics, designers will become familiar with them and be able to apply them in a wide variety of domain-specific scenarios, as has happened with object-oriented design.

3. EXAMPLE

We begin with a simple example, shown in figure 2. The model at the top left contains a class definition labeled "NoisySinewave" and an instance of that class labeled "InstanceOfNoisySinewave." The class definition icon is outlined in light blue to distinguish it visually from an instance. The NoisySinewave class is defined hierarchically by the model on the lower left. It is a subclass of Sinewave, which is the model at the right. NoisySinewave inherits actors, ports, and parameters from Sinewave. The inherited components are outlined with a dashed pink line, indicating visually that they are inherited components. The NoisySinewave class extends the Sinewave class by adding some additional actors, connections, and ports. These additions do not have the dashed pink outline.

The model in figure 2, when executed, produces two signal traces, as shown in the plot at the lower right. One is a simple sine wave and the other is a noisy sine wave. The simple sine wave is generated by the InstanceOfSinewave actor, which is an instance of Sinewave, and the noisy sine wave is generated by the InstanceOfNoisySinewave actor, which is an instance of NoisySinewave, a subclass of Sinewave.

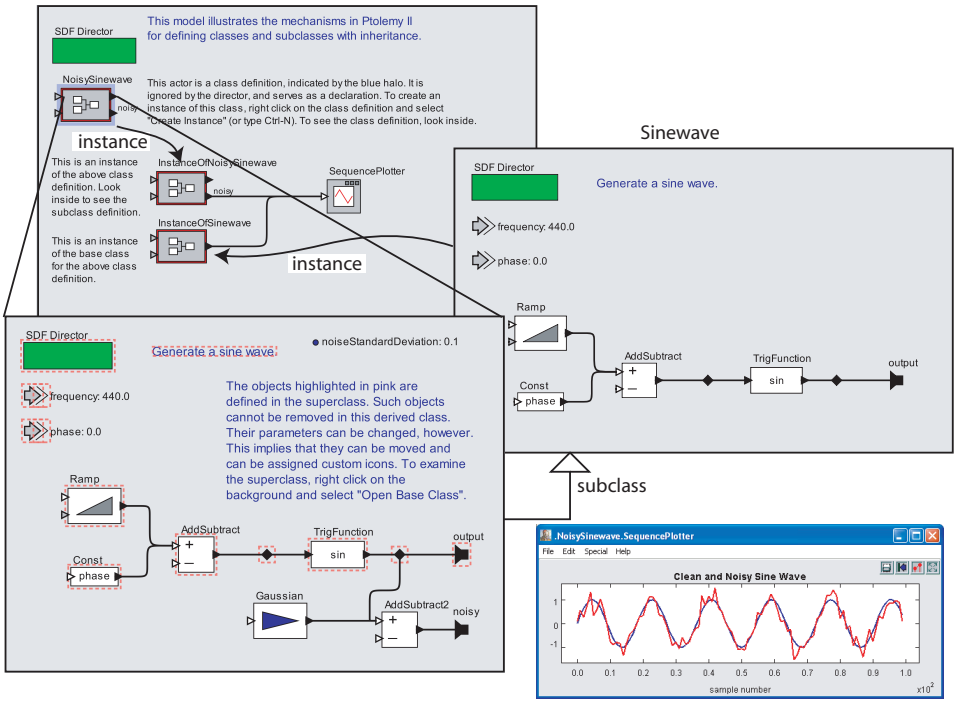

Fig. 2. A simple example of the use of classes in Ptolemy II.

In building this mechanism in Ptolemy II, we had to make a number of decisions that amount to language design decisions. The mechanism we have settled on is the one we explain and attempt to defend in this paper. However, we remind the reader that mechanisms for object-oriented design took some 40 years to mature. While we can expect faster maturity with actor-oriented design (we have learned from the OO experience), it will not be instantaneous. Our design is one of many possibilities, and while we believe it to be effective and aesthetic, we have no doubt that it can be improved. We explain our mechanism informally first, and then give a precise definition. The precise definition is required to fully grasp the subtleties of inner classes.

First, in Ptolemy II, a *model* is a set of actors, ports, attributes, and connections. A model can be viewed as a program with a visual syntax. Each of the grey boxes in figure 2 is a model. A special attribute called a *director* defines the semantics of the model. Each of the models in figure 2 has a director, indicated by the green box at the upper left in the model. For our purposes here, the director is irrelevant, and can be viewed as any other attribute. The visual annotations in the models are also attributes.

In Ptolemy II, any model can be either a class or an instance. A class serves as a prototype for instances. Our mechanism, therefore, is closely related to prototype-based languages (see chapter 3 of [Craig 2001], for example), but with a twist. In order to ensure that the class mechanism operates entirely at the abstract syntax

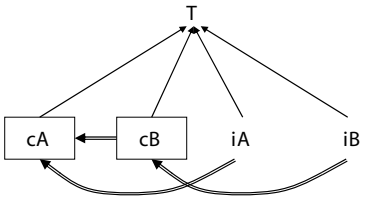

Fig. 3. A model **T** containing four objects, the classes **cA** and **cB** and their instances **iA** and **iB**.

level, classes in Ptolemy II are purely syntactic objects and play no role in the execution of a model. They are not visible to the director, which provides the execution engine. As consequence, Ptolemy II does not permit the ports of a class to be connected to other ports.

A class may be defined in its own file (in which case we call it a *top-level class*) or as a component in a model. The Sinewave class in figure 2 is a top-level class, while NoisySinewave is not. When a class is defined within a model, its definition is in scope at the same level of the hierarchy where it is defined and at all levels below that. This is the same scoping rule that applies to attributes in the Ptolemy II expression language (see [Brooks et al. 2004]). Thus, for example, the model at the upper left in figure 2 contains both the class definition NoisySinewave and the instance InstanceOfNoisySinewave.

A subclass inherits the structure of its base class. Specifically, as we will define formally below, every object (actor, attribute, port or connection) contained by the base class has a corresponding object in the subclass. We refer to this as the *derivation invariant*. The pink dashed outlines in figure 2 surround such "corresponding objects." They provide a visual indication that those objects cannot be removed, since doing so would violate the derivation invariant. However, the subclass can contain new objects and can also change (override) the values of attributes that carry values (we generally refer to attributes that carry values as *parameters*).

Since a model can contain class definitions, and a model can itself be a class definition, we have *inner classes*. This is a significant departure from the prototype mechanism given in [Karsai et al. 2003], where it is (correctly) pointed out that such inner classes create significant complications. In particular, as we will explain below, they create a specialized form of multiple inheritance. Although this is a significant complication, we believe that it is sufficiently disciplined and expressive to be justified.

4. FORMAL STRUCTURE

Figure 3 shows a hierarchy where a top-level model named **T** contains four objects, the classes **cA** and **cB** and their instances **iA** and **iB**. The containment relation is indicated by the solid lines, and *parent* relation is indicated by the double lines. By "parent" we mean either subclassing or instantiation. Thus, **cB** is a subclass of **cA**, while **iB** is an instance of **cB**. The boxes in the figures indicate classes, while the unboxed elements are instances. We require that objects that share the same container have unique labels, and an individual object within a hierarchy may be referenced by a list of labels following the containment hierarchy. Thus, figure 3

contains five objects with full labels **.T**, **.T.cA**, **.T.cB**, **.T.iA**, and **.T.iB**.

4.1 Derivable Objects

Let D be the set of *derivable* objects. These include actors, models (which are actors), attributes and ports. The *container relation* is a partial function

$$c\colon D \to D$$

where $c(x) = y$ means that x is contained by y. Since this relation is a partial function, a derivable object can have at most one container. When $c(x) = y$ we can also write $(x, y) \in c$, and we say "x is contained by y."

Let c^+ be the transitive closure of the container relation. That is, $(x, y) \in c^+$ if $(x, y) \in c$ or $(c(x), y) \in c^+$. We disallow circular containment, so if $(x, y) \in c^+$ then $(y, x) \notin c^+$ (that is, c^+ is *anti-symmetric*). Since c^+ is also *irreflexive* ($(x, x) \notin c^+$) and *transitive* ($(x, y) \in c^+$ and $(y, z) \in c^+ \Rightarrow (x, z) \in c^+$), then (D, c^+) is a strict partially ordered set (*strict poset*).

The *parent* relation is a partial function

$$p\colon D \to D$$

where $p(x) = y$ means that either x is a subclass of y or x is an instance of y. In either case, we refer to y as the *parent* and x as the *child*. Since this is a partial function, a derivable object may have at most one parent. This would seem to rule out multiple inheritance, but as we will see, inner classes provide a disciplined form of multiple inheritance. When $p(x) = y$ we can also write $(x, y) \in p$ and say that "x is a child of y."

Let p^+ be the transitive closure of the parent relation, just as with c^+. Again, we disallow circular parent relations, so (D, p^+) is a strict poset.

(D, c^+) and (D, p^+) are each strict posets. We impose a key additional constraint, which is that if $(x, y) \in c^+$ then $(x, y) \notin p^+$ and $(y, x) \notin p^+$. That is, if x is contained directly or indirectly by y, then it cannot be a child of y, directly or indirectly, nor can y be its child, directly or indirectly. Correspondingly, if $(x, y) \in p^+$ then $(x, y) \notin c^+$ and $(y, x) \notin c^+$. Following Davis [John Davis 2000], we refer to (D, c^+, p^+) as a *doubly nested diposet*.

Let L be the set of all labels. The *labeling function* is

$$l\colon D \to L$$

where we require that if $c(x) = c(y)$ and $x \neq y$, then $l(x) \neq l(y)$. The *full label* of x is the sequence of labels from the top-level container of x (the unique y such that $(x, y) \in c^+$ and $c(y)$ is undefined) to the label of x. We can write this sequence separated by periods. For the example in figure 3, using the full labels, $(\textbf{.T.cA}, \textbf{.T}) \in c$ and $(\textbf{.T.iB}, \textbf{.T.cB}) \in p$. Moreover, $(\textbf{.T.iB}, \textbf{.T.cA}) \in p^+$. Note that full labels need not be unique in D, so by themselves, they do not provide the identify of an object in D.

4.2 Derived Relation

The key to our notion of inner classes is the *derived relation* $d \subset D \times D$, defined to be the least relation such that $(x, y) \in d$ implies that either

$$(x, y) \in p^+$$

or

$$(c(x), c(y)) \in d \text{ and } l(x) = l(y).$$

That is, x is derived from y if either of the following is true:

(1) x is a child of y (directly or indirectly) or
(2) x and y have the same label and the container of x is derived from the container of y.

We say that y is *implied by* z in D if $(y, z) \in d$ and $(y, z) \notin p^+$. If y is implied by z, then it is not necessary to represent y explicitly in a persistent representation of the model (unless it *overrides* z in some way, as we will discuss). This is because the derivation invariant alone implies the existence of y, and hence, any tool that reads the persistent representation and constructs the model such that the derivation invariant is true will construct y simply as a consequence of constructing z. The ability to omit components from a persistent representation can be very helpful. Even for some relatively simple and practical Ptolemy II models, we have found that the XML representation of the model could shrink by a factor of 10 or more. However, it is imperative that we accurately exclude only those components that are implied. This turns out to be particularly challenging when persistent representations need to be generated for only a portion of a model (for example, in copy-and-paste functionality, where a portion of a model is copied and then pasted elsewhere into another model). To get this right, we need to formalize the derivation invariant.

4.3 Derivation Invariant

Informally, the derivation invariant states that if x is derived from y, then x has at least the structure of y. We now state the *derivation invariant* formally. If $(x, y) \in d$, then the following property holds:

> For all z where $c(z) = y$, there exists a z' where $c(z') = x$ and $l(z) = l(z')$ and either
> (1) $p(z)$ and $p(z')$ are undefined, or
> (2) $(p(z), p(z')) \in d$, or
> (3) $p(z) = p(z')$ and both $(p(z), y) \notin c^+$ and $(p(z'), x) \notin c^+$.

The first alternative states that if x is derived from y, then for every z contained by y there is a corresponding z' contained by x with the same label. Clearly $(z', z) \in d$. This means that x contains (deeply) an object corresponding to each object contained (deeply) by y.

The second alternative states that parent-child relationships in y are mirrored in x. That is, for every z contained by y and z' contained by x such that $(z', z) \in d$, z has a parent if and only if z' has a parent. If they both have parents, then either the parent of z' is derived from the parent of z, or they both have the same parent, and that parent is not (deeply) contained by either x or y.

The consequence is that if x is derived from y, then x has the same structure as y, in the sense that it contains objects "corresponding to" those in y, and that any parent relations that are entirely within y (i.e., both parent and child are (deeply) contained by y) have corresponding parent relations that are entirely within x.

That is, it "inherits" the structure of y. A similar invariant can be defined for connections between ports, but we leave that as an exercise.

The third alternative states that z and z' have the same parent, but that parent is not (deeply) contained by either y or x. That is, the parent of both z and its implied z' is external to both y and x. In Ptolemy II, for example, that parent might be a top-level class.

In practice, derived objects may have more similarity than is implied by simply having the same "label." Formally, since we have not given any semantics to a label, this constraint can emcompass any measure of similarity that we wish. In Ptolemy II, for example, derived components are required to be represented by instances of the same Java class. That is, if x is derived from y, then x must be an instance of the same Java class (or a derived Java class) that y is an instance of. This binds the AO design structure to the OO design structure in very useful ways.

4.4 Persistent Representation

Suppose we are given a model that is a top-level component z that contains various other components with various parent-child relationships. Suppose further that we wish to construct a persistent representation of a particular component y and all of its contents within this model (for example, to perform a copy-and-paste operation within a model editor). A key question arises: what portion of the model should be represented? We do not wish to include anything that is not contained (deeply) by y (thus, for example, if something deeply contained by y is an instance of a class that is not defined within y, then the representation can only be pasted into a context where that class definition is in scope). However, we also do not need to contain any component that is deeply contained by y but is implied by some other component that is also deeply contained by y. Thus, we seek a minimal subset of the model that is sufficient to reconstruct y.

Define the down set function on the container relation by

$$\downarrow_c : D \rightarrow P(D),$$

where $P(D)$ is the powerset of D, such that

$$x \in \downarrow_c (y) \iff (x, y) \in c^+ \text{ or } x = y.$$

A persistent representation of y does not need to include anything that is not in $\downarrow_c (y)$.

Define the up set function on the parent relation by

$$\uparrow^p : D \rightarrow P(D)$$

such that

$$x \in \uparrow^p (y) \iff (y, x) \in p^+ \text{ or } x = y.$$

Thus, if y itself is an instance of a class x, then $x \in \uparrow^p (y)$.

Both the down set and up set functions can be lifted and applied to sets. For example, we define

$$\downarrow_c : P(D) \rightarrow P(D),$$

so that for all $Y \subset D$,

$$\downarrow_c (Y) = \cup_{y \in Y} \downarrow_c (y)$$

and similarly for the up set.

As a shorthand, define the *upset downset* function by

$$\updownarrow_c^p \colon D \to P(D)$$

such that

$$\updownarrow_c^p (y) = \downarrow_c \uparrow^p \downarrow_c (y).$$

The significance of the upset downset is that it contains all the information that is needed to reconstruct the model (deeply) contained by y. A key property is that this subset of D is internally self-consistent in that it satisfies the derivation invariant when the derived relation is restricted to this subset. That is, define the (restricted) derived relation $d_y \subset \updownarrow_c^p (y) \times \updownarrow_c^p (y)$ such that

$$x \in \updownarrow_c^p (y) \text{ and } x' \in \updownarrow_c^p (y) \text{ and } (x, x') \in d \Rightarrow (x, x') \in d_y.$$

Clearly, $d_y \subseteq d$. Then restricted to $\updownarrow_c^p (y)$ and d_y, the derivation invariant still holds. Specifically, if $(x, x') \in d_y$ then it must be true that

$$\forall z' \in \updownarrow_c^p (y) \text{ where } c(z') = x',$$

$$\exists z \in \updownarrow_c^p (y) \text{ where } c(z) = x \text{ and } l(z) = l(z').$$

It is now obvious that a persistent representation of y and its contained objects needs to include only those components in $\downarrow_c (y)$ that are not implied in $\updownarrow_c^p (y)$. To see this, simply note that if we are given this persisent representation and $\updownarrow_c^p (y) \setminus \downarrow_c (y)$, where the backslash denotes set subtraction, then we can generate $\downarrow_c (y)$.

The software implementation of this mechanism in Ptolemy II makes the following observation when constructing a persistent representation of y and its contents. For each component deeply contained by y, it is sufficient to know how far above it in the hierarchy is a parent-child relationship that implies the object. If there is no such parent-child relationship, then clearly the component needs to be represented explicitly. If the parent-child relationship is above y in the hierarchy, then again the component needs to be explicitly represented. If the parent-child relationship is at y (i.e. y is the child) or at some component (deeply) contained by y, then the component is implied, and it does not need to be represented.

Of course, if a component (deeply) contained by y is overridden in some way, then we may need to represent it anyway. We next discuss the override mechanisms.

4.5 Dynamic Structure

We assume that the structure of a model can change during execution of the model. That is, new instances can be created, new subclasses can be defined, and new classes can be defined. Each of these changes will represent a change to the key relations c and p. We assume that such changes are atomic and sequential, and that after every change, the derivation invariant remains true. It is, of course, a challenge in the design of the Ptolemy II software to ensure that this is true, particularly since the software system is intrinsically highly concurrent.

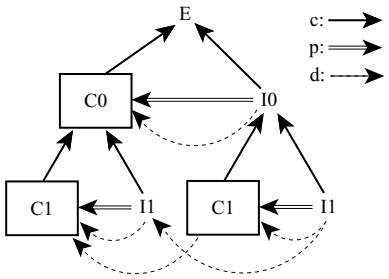

Fig. 4. A model **E** with inner classes.

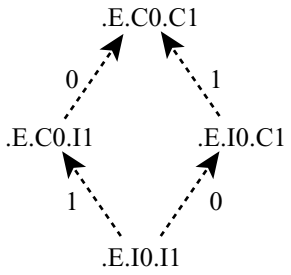

Fig. 5. Derived relation among **.E.C0.C1**, **.E.C0.I1**, **.E.I0.C1**, and **.E.I0.I1**.

4.6 Subclassing

We can use the derived relation to cleanly define overriding, which allows for AO subclassing. In particular, if x is derived from y, the derivation invariant does not prevent x from containing *additional* objects that have no corresponding object in y. The NoisySinewave subclass in figure 2 contains just such additional objects.

Certain objects in a model have *values*. For example, parameters of an actor have values. Let the *valuation* be a function

$$v\colon D \to V$$

where V is a set of values that includes a special element meaning "undefined" or "has no value." We will assume that v can also change during execution of the model, but as with the changes to c and p, the changes are atomic and sequential. A key issue is to determine whether $(x, y) \in d$ implies that $v(x) = v(y)$. This question relates to inheritance, but is somewhat more complicated than the structural inheritance described above. In particular, a subclass may override the value of an object, and that override may shadow further derived objects. It is precisely this complication that lead the authors of [Karsai et al. 2003] to disallow inner classes. We have taken a more aggressive stand, which is to allow subclasses and to give a clean semantics to overriding. This choice has proven useful in a number of practical Ptolemy II designs. It certainly enriches the model, and makes it much more modular, since classes can locally contain locally class definitions.

Figure 4 shows a model with inner classes. The derived relation is shown with dashed lines. This relation alone, which relates **.E.C0.C1**, **.E.C0.I1**, **.E.I0.C1**,

and **.E.I0.I1**, is extracted in figure 5. The numbers annotating the arrows are given by a partial function h that we call the *derivation depth*,

$$h\colon D \times D \to N_0$$

where $N_0 = \{0, 1, 2, \ldots\}$ is the natural numbers. $h(x, y)$ is undefined if $(x, y) \notin d$, or if $(x, y) \in d$ but the derived relationship between x and y is not established from an immediate parent-child relationship. $h(x, y) = i$ if the derived relationship between x and y is established from an immediate parent-child relationship i levels above x and y in the containment hierarchy. **.E.I0.I1** inherits the deeply contained objects and their values, if any, from all other objects in figure 5. When a contained object has different values in **.E.C0.C1**, **.E.C0.I1**, and **.E.I0.C1**, which should **.E.I0.I1** inherit the value from? This is similar to problems with multiple inheritance in object-oriented programming.

Generally, given any $x \in D$, x inherits from all objects in $I(x) \triangleq \uparrow^d (x) \setminus \{x\}$. Our approach is to define a linear order on $I(x)$ to prioritize the inheritance. The order is constructed from the container relation and the parent relation. It is consistent with the derived relation. For example, both **.E.C0.I1** and **.E.I0.C1** have higher priority than **.E.C0.C1**. The order favors local definitions. For example, **.E.I0.C1** has higher priority than **.E.C0.I1**.

To compute the linear order, define a partial function (override strength)

$$s\colon D \times D \to [N_0]$$

$[N_0]$ is the set of lists of numbers from N_0. $s(x, y)$ is undefined if $(x, y) \notin d$. For any $(x, y) \in d$, let $S(x, y)$ be the multiset of h values anotated on a path from x to y in the graph of the derived relation d. For example, in figure 5, $S(\textbf{.E.I0.I1}, \textbf{.E.C0.C1}) = \{0, 1\}$. Different paths from x to y in the graph of d yield the same multiset. From $S(x, y)$, let

$$s(x, y) = [a_0, \ldots, a_{\max S(x,y)}]$$

where a_k is the number of k's in $S(x, y)$. For example,

$$\begin{aligned} s(\textbf{.E.I0.I1}, \textbf{.E.C0.C1}) &= [1, 1] \\ s(\textbf{.E.I0.I1}, \textbf{.E.C0.I1}) &= [0, 1] \\ s(\textbf{.E.I0.I1}, \textbf{.E.I0.C1}) &= [1] \end{aligned}$$

As a special case, let $s(x, x) = [0]$ for all $x \in D$. Except for this special case, $s(x, y)$ always has its final value greater than zero.

Define a *priority* order $>_p$ on $[N_0]$ as, for given $l_1 = [a_0, \ldots, a_m]$ and $l_2 = [b_0, \ldots, b_n]$, $l_1 >_p l_2$ if $m < n$. If $m = n$, then there is a $k \leq m$ such that $b_i = a_i$ for all i, $k < i \leq m$, and we define $l_1 >_p l_2$ if $a_k < b_k$. That is, we compare the latest elements of l_1 and l_2 that differ. In all other circumstances, unless $l_2 = l_1$, we define $l_2 >_p l_1$. For example,

$$[1] >_p [0, 1] >_p [1, 1]$$

The relation $>_p$ is a total order on $[N_0]$. We use this order to determine whether one propagation path is "closer" than another. Paths with higher priority will dominate those with lower priority when setting values.

Let *override* be a partial function

$$r\colon D \to [N_0].$$

For each $x \in D$, $r(d)$ will specify the priority of the path that has determined the current value of x, if there is such a value and path. The definition of r changes as the function v changes, of course, because it changes when values change. In particular, initially, r is undefined for all $x \in D$. When a value is set for x, the value may propagate to derived objects. We will use r to track whether a value was set directly or via propagation, and if it was via propagation, then along what path in the derived relation. Specifically, we define $r(x) = [0]$ if its value is set directly. We define $r(x) = s(x, y)$ if its value is set due to propagation from y.

We can now determine whether a change to the value of y should propagate to x. If $(x, y) \notin d$, then it should not. If $(x, y) \in d$ but $r(x)$ is undefined, then it should, so that $v(x) = v(y)$. If $(x, y) \in d$ but $r(x) = [0]$, then it should not, because x has been previously set directly. If $(x, y) \in d$ but $s(x, y) >_p r(x)$, then it should, because x has been previously set, but the propagation occurred higher up in the containment hierarchy or more remote over the parent-child relation than the current one. † If propagation occurs, then together with the change in in $v(x)$, we must define $r(x) = s(x, y)$.

This multiple inheritance mechanism has been implemented in Ptolemy II, which is open source and can be downloaded from http://ptolemy.org. The download includes several examples that make extensive use of actor-oriented classes, and also includes an extensive test suite that tests every feature of the implementation.

5. CONCLUSION

We have argued that actor-oriented design can benefit from abstraction and modularity mechanisms similar to what has been developed in object-oriented languages. We have given a preliminary formalism that provides a structure for classes and inheritance. There are a number of issues that have been left out, but that amount to fairly obvious extensions. For example, our mechanism permits subclasses to have additional ports, but does it make sense for a subclass to have additional *input* ports? If this is allowed, then a subclass cannot be viewed as an instance of the base class because required inputs will not be provided. A co/contravariance similar to type systems in procedural languages is required. A similar issue applies to the types of parameters and ports, which have not been discussed. Finally, once mechanisms like this have been defined, it becomes possible to establish a clear separation between interfaces and implementations.

REFERENCES

AGHA, G. 1986. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA.

AGHA, G. 1990. Concurrent object-oriented programming. *Communications of the ACM 33,* 9, 125–140.

†It is language design issue what the direction of this inequality should be. We have chosen this direction because it seems that propagations that are more local should take precedence over ones that are more global. Any resolution of this question, however, must be unique.

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

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

Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., and Zheng, H. 2004. Heterogeneous concurrent modeling and design in Java. Tech. Rep. Technical Memorandum UCB/ERL M04/27, University of California. July 29.

Buck, J. T., Ha, S., Lee, E. A., and Messerschmitt, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development" 4*, 155–182.

Craig, I. 2001. *The Interpretation of Object-Oriented Programming Languages*. Springer-Verlag.

Dennis, J. B. 1974. First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science. Probably the seminal dataflow paper. It has a merge operator (same as my select) and a T-gate and F-gate (the two sides of my switch). The notion of colored tokens is used to distinguish among concurrent activations of the same dataflow procedure. The static-dataflow restriction that only one token of any color can appear at one time on any arc is used throughout. A global heap is proposed to handle data structure manipulation. Actors select, append, and the predicate exists are introduced to operate on data structures. An apply actor is used for function application. The paper concludes with a list of open problems. The first is the parallel for, which is solved by the SDF method of multiple token production. The second is determinate procedures with memory: a problem solved with feedback loops and delays. The third is data abstraction. The fourth is nondeterminate computation (see Arvind and Brock).

Eker, J. and Janneck, J. W. 2003. Cal language report: Specification of the cal actor language. Tech. Rep. Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA. December 1.

Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. 2003. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE 91,* 2, 127–144.

Esser, R. 1996. Ph.d. thesis. Ph.D. thesis, ETH.

Esser, R. and Janneck, J. W. 2001. A framework for defining domain-specific visual languages. In *Workshop on Domain Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA-2001*. Tampa Bay, Florida, USA.

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

Goessler, G. and Sangiovanni-Vincentelli, A. 2002. Compositional modeling in metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*. Springer-Verlag, Grenoble, France.

Gssler, G. and Sifakis, J. 2005. Composition for component-based modeling. *Science of Computer Programming 55*.

Hewitt, C. 1977. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence 8,* 3, 323363.

Jackson, E. K. and Sztipanovits, J. 2006. Towards a formal foundation for domain specific modeling languages. In *EMSOFT*. ACM Press, Seoul, Korea, 53–62.

John Davis, I. 2000. Ph.d. thesis. Ph.D. thesis, UC Berkeley.

Johnson, S. C. 1994. Objecting to objects. In *USENIX Winter 1994 Technical Conference Proceedings*. San Francisco, California.

Karsai, G. 1995. A configurable visual programming environment: A tool for domain-specific programming. *IEEE Computer*, 36–44.

Karsai, G., Maroti, M., Ldeczi, k., Gray, J., and Sztipanovits, J. 2003. Type hierarchies and composition in modeling and meta-modeling languages. *IEEE Transactions on Control System Technology to appear*.

Keutzer, K., Malik, S., Newton, A. R., Rabaey, J., and Sangiovanni-Vincentelli, A. 2000. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19,* 12.

Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. 2001. Composing domain-specific design environments. *IEEE Computer*, 44–51.

Lee, E. A. 2006. The problem with threads. *Computer 39,* 5, 33–42.

Lee, E. A. and Neuendorffer, S. 2000. Moml - a modeling markup language in xml. Tech. Rep. UCB/ERL M00/12, UC Berkeley. March 14.

Lee, E. A., Neuendorffer, S., and Wirthlin, M. J. 2003. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers 12,* 3, 231–260.

Liao, S., Tjiang, S., and Gupta, R. 1997. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *Design Automation Conference*. ACM, Inc., Anaheim, CA.

Lynch, N., Segala, R., Vaandrager, F., and Weinberg, H. 1996. Hybrid I/O automata. In *Hybrid Systems III*, R. Alur, T. Henzinger, and E. Sontag, Eds. Vol. LNCS 1066. Springer-Verlag, 496–510.

Lynch, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Reppy, J. H. 1991. Cml: A higher-order concurrent language. *SIGPLAN Notices 26,* 6, 293–305.

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

Stewart, D. B., Volpe, R., and Khosla, P. 1997. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. on Software Engineering 23,* 12, 759–776.

Sutherland, I. E. 1963. Sketchpad - a man-machine graphical communication system. Tech. Rep. Technical Report 296, MIT Lincoln Laboratory. January.

Sutherland, W. R. 1966. Ph.d. thesis. Ph.D. thesis, MIT.

Thies, W., Karczmarek, M., and Amarasinghe, S. 2002. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*. Vol. LNCS 2304. Springer-Verlag, Grenoble, France.