Invited paper: Conference on Formal Methods and Models for Codesign
(MEMOCODE) June 22-25, 2004

Classes and Subclasses in Actor-Oriented Design

Extended Abstract

Edward Lee and Stephen Neuendorffer
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.

Abstract

Actor-oriented languages provide a component composition methodology that emphasizes concurrency. The interfaces to actors are parameters and ports (vs. members and methods in object-oriented languages). Actors interact with one another through their ports via a messaging schema that can follow any of several concurrent semantics (vs. procedure calls, with prevail in OO languages). Domain-specific actor-oriented languages and frameworks are common (e.g. Simulink, LabVIEW, and many others). However, they lack many of the modularity and abstraction mechanisms that programmers have become accustomed to in OO languages, such as classes, inheritance, interfaces, and polymorphism. This extended abstract shows the form that such mechanisms might take in AO languages. A prototype of these mechanisms realized in Ptolemy II is described.

1 Introduction

Actor-oriented languages and frameworks are growing in popularity, particularly for embedded software and domain-specific design. It is conceivable that actor-oriented languages could eventually have as much impact in computing as object-oriented languages have had. It has the potential to deliver similar scaling and productivity gains, on top of those that have been delivered by OO languages. It can coexist with and complement OO languages, but offers better mechanisms for managing concurrency and real time that extend well beyond those built into OO languages. As a consequence, AO languages are well-suited to embedded software and distributed real-time systems.

Actor-oriented design has been around since at least 1966, when Bert Sutherland used one of the first acknowledged object-oriented frameworks (Sketchpad [26]), created by his brother Ivan Sutherland, to build the first actor-oriented programming language (which had a visual syntax) [27]. 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 [5].

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 [5], 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, can be adapted to operate at the level of AO design. We describe preliminary implementations of these mechanisms in Ptolemy II and illustrate the mechanisms with a simple example.

2 Actor-Oriented Design

Actor-oriented design is a component methodology that has proven effective for 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

This research 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.

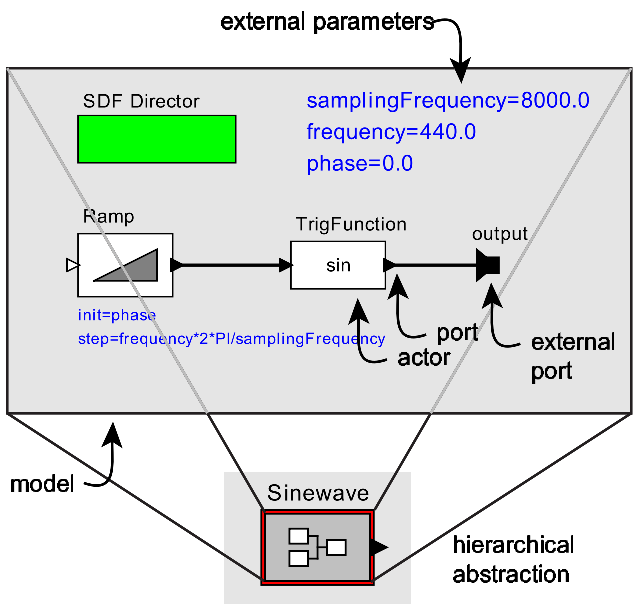


Figure 1. Illustration of an actor-oriented model (above) and its hierarchical abstraction (below).

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 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 syntax can be represented concretely in several ways, such as graphically, as in figure 1, in XML [20], or in a program designed to a specific object-oriented API. Ptolemy II [10] offers all three alternatives.

2.1 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*. 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 [13]. Many such systems have visual syntaxes and are often viewed more as modeling tools than as programming languages; in this extended abstract, 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 extended abstract 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.

3 Related Work

3.1 Software Components

Prevailing software component architectures such as CORBA, DCOM, and Java Beans, are deeply rooted in the procedural semantics of the dominant object-oriented languages C++ and Java. In such procedural semantics, concurrency is managed using threads, monitors and semaphores, a notoriously difficult approach. Conventions that ensure deadlock avoidance, such as acquisition of locks in a fixed order (see for example [19]), are not supported by the languages (nothing about a method signature declares what locks it will acquire, for instance). As a consequence, these conventions are difficult to apply in practice, and seemingly innocent changes to code can create disastrous failures such as deadlock.

As a result, it is difficult to treat objects in object-oriented languages as components 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. Moreover, actor-oriented abstractions can embrace time and concurrency, and therefore match much better the modeling of embedded systems, which are intrinsically concurrent.

3.2 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 [15]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1, 2, 3, 4]. 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 [8].

A number of more recent efforts adopt an actor-oriented approach. ROOM (Real-time Object-Oriented Modeling [24]) 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 [25], I/O automata [22] and hybrid I/O automata [21], Moses [12], Polis and Metropolis [14], Ptolemy [5] and Ptolemy II [10] all emphasize actor orientation. Languages for designing actors are a current research topic; for example StreamIT [28], which calls actors "filters," and Cal [9] are languages for designing hardware and software components that interact with dataflow semantics.

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

3.3 Prototypes and Classes in Actor-Oriented Languages

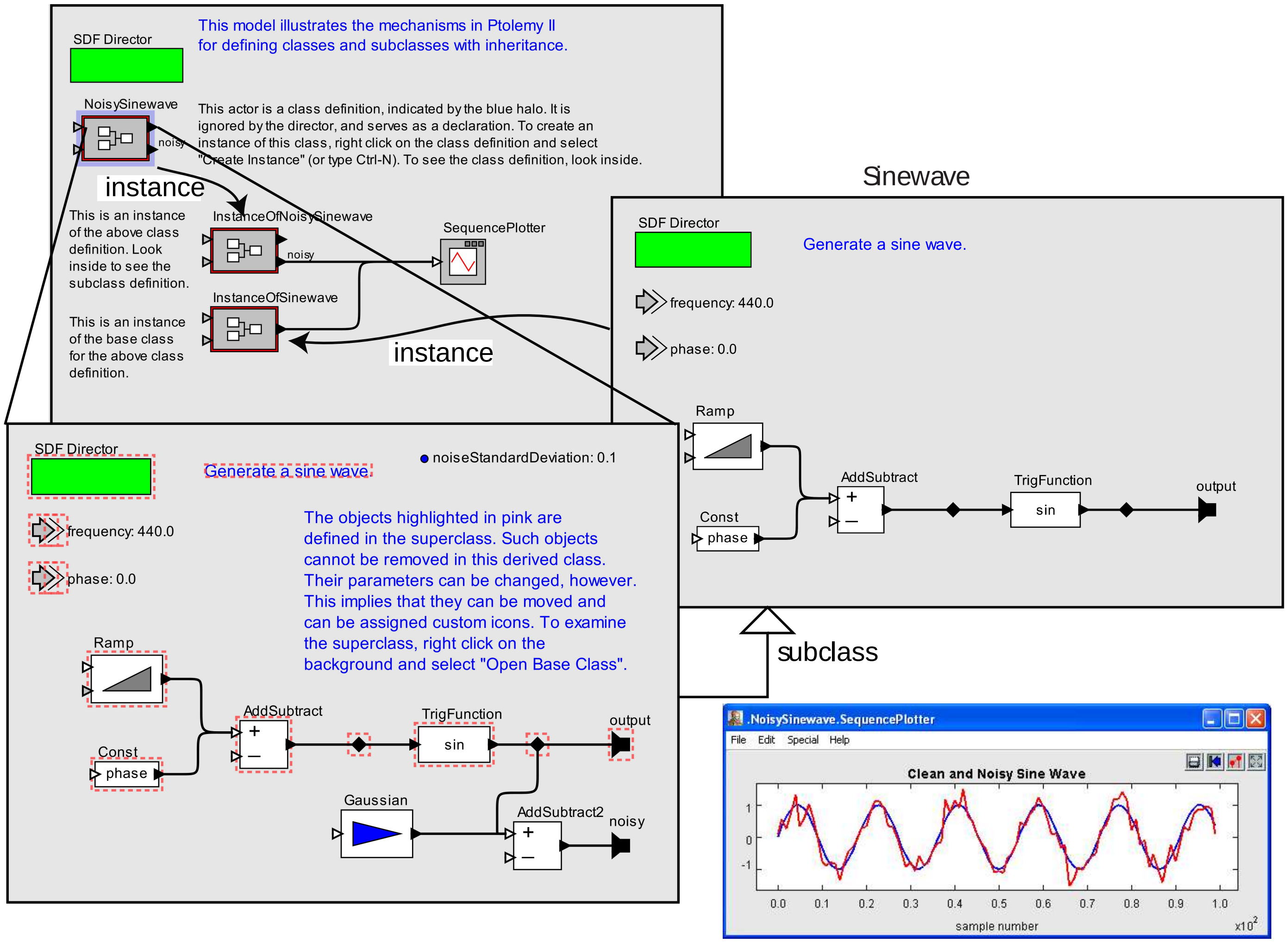
This paper is about extending actor-oriented design techniques with modularity mechanisms like those in OO languages. A number of interesting experiments in this direction have been performed by others. The GME system from Vanderbilt [17] has been extended to support actor-oriented prototypes [18]. This 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 [11], from ETH builds in an OO notion of classes into a design environment based on time Petri nets. Concurrent ML [23], 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) [24], 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.

4 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



This model illustrates the mechanisms in Ptolemy II for defining classes and subclasses with inheritance.

This actor is a class definition, indicated by the blue halo. It is ignored by the director, and serves as a declaration. To create an instance of this class, right click on the class definition and select "Create Instance" (or type Ctrl-N). To see the class definition, look inside.

This is an instance of the above class definition. Look inside to see the subclass definition.

This is an instance of the base class for the above class definition.

Generate a sine wave.

Generate a sine wave.

The objects highlighted in pink are defined in the superclass. Such objects cannot be removed in this derived class. Their parameters can be changed, however. This implies that they can be moved and can be assigned custom icons. To examine the superclass, right click on the background and select "Open Base Class".

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

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.

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. We explain this mechanism informally first, and then in the next section 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 [6], for example), but with a twist. In order to ensure that the class mechanism operates entirely at the abstract syntax 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

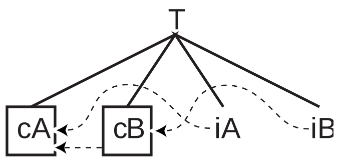


Figure 3. A model T containing four objects, the classes cA and cB and their instances iA and iB.

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 [16]). 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 [18], 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.

5 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 dashed 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 must have unique names, and an individual object within a hierarchy may be referenced by its *full name*, which is a dotted name showing the containment. Thus, figure 3 contains five objects with full names **.T**, **.T.cA**, **.T.cB**, **.T.iA**, and **.T.iB**.

5.1 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 $(x, y) \in c$.

Let S be the set of all names. The *naming function* is

$$n\colon D \to S$$

where we require that if $c(x) = c(y)$, then $n(x) \neq n(y)$. The full name of an object is a sequence of names, a member of the set S^*.

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 limited form of multiple inheritance. When $p(x) = y$ we can also write $(x, y) \in p$.

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 [7], we refer to (D, c^+, p^+) as a *doubly nested diposet*.

For the example in figure 3, $(\textbf{.T.cA}, \textbf{T}) \in c$ and $(\textbf{.T.iB}, \textbf{.T.cB}) \in p$. Moreover, $(\textbf{.T.iB}, \textbf{.T.cA}) \in p^+$.

5.2 Derived Relation

The key to our notion of inner classes is the *derived relation* $d \in D \times D$ defined as follows. The pair $(x, y) \in d$ if either $(x, y) \in p^+$ or $n(x) = n(y)$ and $(c(x), c(y)) \in d$. 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 name and the container of x is derived from the container of y.

5.3 Derivation Invariant

We can now state the *derivation invariant* formally. If $(x, y) \in d$, then for all z where $c(z) = y$, there exists a z' where $c(z') = x$ and $n(z) = n(z')$. That is, if x is derived from y, then for every z contained by y there is a corresponding z' contained by x with the same name. It is now trivial to see that $(z', z) \in d$. The consequence is that x is derived from y, then x has the same containment structure as y, in the sense that it contains objects "corresponding to" those in y. 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.

So far, this correspondence involves little more than mirroring the hierarchy structure with objects that have the same names. In practice, we will constrain these objects to have more than the same names. In Ptolemy II, for example, these objects must be instances of the same Java class. That is, if x is derived from y, then x must be an instance of the same class (or a derived class) that y is an instance of. This binds the AO design structure to the OO design structure in very useful ways.

5.4 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.

5.5 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

$$v : D \to V$$

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 [18] 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 stand is somewhat speculative, in that we are not sure that it will yield practical design value. But it certainly enriches the model, and makes it much more modular, since classes can locally contain locally used class definitions.

The *depth* relation is a partial function

$$h : D \times D \to N$$

where $N = \{1, 2, \ldots\}$ is the set of natural numbers and

$$h(x,y) = \begin{cases} 1 & \text{if } (x,y) \in p^+ \\ 1 + h(c(x), c(y)) & \text{if } (x,y) \notin p^+ \text{ and } (x,y) \in d \\ \text{undefined} & \text{otherwise} \end{cases}$$

The depth function tells us how far up the hierarchy the parent-child relationship is that induces a given derived relation. We will use to determine whether or not a change to a value should propagate to a particular derived object.

Let *override* be a partial function

$$r : D \to N_0,$$

where $N_0 = \{0, 1, 2, \ldots\} = N \cup \{0\}$. The definition of r changes as the function v changes. 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 at what level of the hierarchy is the parent-child relation that induces the propagation. Specifically, we define $r(x) = 0$ if its value is set directly. We define $r(x) = h(x, y)$ 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 $h(x, y) \geq r(x)$, then it should, because x has been previously set, but the propagation occurred further down in the hierarchy than the current one.[†] If propagation occurs, then together with the change in in $v(x)$, we must define $r(x) = h(x, y)$.

6 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

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

[2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–140, 1990.

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

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

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

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

[†] It is language design issue what the direction of this inequality should be. We have (tentatively) chosen this direction because it seems that propagations that are more global should take precedence over ones that are more local, but either choice is arguable.

[7] J. Davis II. *Order and Containment in Concurrent System Design*. Ph.d. thesis, UC Berkeley, 2000.

[8] J. B. Dennis. First version data flow procedure language. Technical Report MAC TM61, MIT Laboratory for Computer Science, 1974.

[9] J. Eker and J. W. Janneck. Cal language report: Specification of the CAL actor language. Technical Report Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA, December 1 2003.

[10] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2), 2003.

[11] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. Ph.d. thesis, ETH, 1996.

[12] R. Esser and J. W. Janneck. 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, 2001.

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

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

[15] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323363, 1977.

[16] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, and H. Zheng. Heterogeneous concurrent modeling and design in java. Technical Report Technical Memorandum UCB/ERL M03/27, University of California, July 16, 2003 2003.

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

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

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

[20] E. A. Lee and S. Neuendorffer. MoML - a modeling markup language in XML. Technical Report UCB/ERL M00/12, UC Berkeley, March 14 2000.

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

[22] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[23] J. H. Reppy. CML: A higher-order concurrent language. *SIGPLAN Notices*, 26(6):293–305, 1991.

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

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

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

[27] W. R. Sutherland. *The On-Line Graphical Specificatoin of Computer Procedures*. Ph.d. thesis, MIT, 1966.

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