System-Level Types for Component-Based Design

Edward A. Lee and Yuhong Xiong
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
{eal, yuhong}@eecs.berkeley.edu

Abstract. We present a framework to extend the concept of type systems in programming languages to capture the dynamic interaction in component-based design, such as the communication protocols between components. In our system, the interaction types and the dynamic behavior of components are defined using interface automata - an automata-based formalism. Type checking, which checks the compatibility of a component with a certain interaction type, is conducted through automata composition. Our type system is polymorphic in that a component may be compatible with more than one interaction type. We show that a subtyping relation exists among various interaction types and this relation can be described using a partial order. This system-level type order can be used to facilitate the design of polymorphic components and simplify type checking. In addition to static type checking, we also propose to extend the use of interface automata to the on-line reflection of component states and to run-time type checking. We illustrate our framework using a component-based design environment, Ptolemy II, and discuss the trade-offs in the design of system-level type systems.

1 Introduction

Type systems are one of the most successful formal methods in software design. Modern polymorphic type systems, with their early error detection capabilities and the support for software reuse, have led to considerable improvements in development productivity and software quality.

For embedded systems, component-based design is established as an important approach to handle complexity. In this area, type systems can be used to greatly improve the quality of design environment. Fundamentally, a type system detects mismatch at component interfaces and ensures component compatibility. Interface mismatch can happen at (at least) two different levels. One is the data type level. For example, if a component expects to receive an integer at its input, but another component sends it a string, then the first component may not be able to function correctly. Many type system techniques in general purpose languages can be applied effectively to ensure compatibility at this level (see [15] and the references therein). The other level of mismatch is the dynamic interaction behavior, such as the communication protocol the components use to exchange data. Since embedded systems often have many concurrent computational activities and mix widely differing operations, components may follow widely different communication protocols. For example, some might use synchronous interaction (rendezvous) while others use asynchronous message passing (see [8] for many more examples). So far, most type system research for component-

First Workshop on Embedded Software, EMSOFT2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001.

based design, as well as for general purpose languages, concentrates on data types, and leaves the checking of dynamic behavior to other techniques.

In this paper, we extend the concept of type systems to capture the dynamic aspects of component interaction. We call the result *system-level types* [10]. In our approach, different interaction types and the dynamic behavior of components are described by automata, and type checking is conducted through automata composition. In this paper, we choose a particular automata model called *interface automata* [4] to define types. The particular strength of interface automata is their composition semantics.

Traditionally, automata models are used to perform model checking at design time. Here, our emphasis is not on model checking to verify arbitrary user code, but rather on compatibility of the composition of pre-defined types. As such, the scalability of the methods is much less an issue, since the size of the automata in question is fixed. We also propose to extend the use of automata to on-line reflection of component state, and to do run-time type checking. To explore these concepts, we have built an experimental platform based the Ptolemy II component-based design environment [3].

We have found that the design of system-level types shares the same goals and trade-offs with the design of a data-level type system. At the data level, research has been driven to a large degree by the desire to combine the flexibility of dynamically typed languages with the security and early error-detection potential of statically typed languages [13]. As mentioned earlier, modern polymorphic type systems have achieved this goal to a large extend. At the system-level, type systems should also be polymorphic to support component reuse while ensuring component compatibility.

In programming languages, there are several kinds of polymorphism. In [1], Cardelli and Wegner distinguished two broad kinds of polymorphism: *universal* and *ad hoc* polymorphism. Universal polymorphism is further divided into *parametric* and *inclusion* polymorphism. Parametric polymorphism is obtained when a function works uniformly on a range of types. Inclusion polymorphism appears in object oriented languages when a subclass can be used in place of a superclass. Ad hoc polymorphism is also further divided into overloading and coercion. In systems with subtyping and coercion, types naturally form a partial order [2]. For example, in object-oriented languages, the partial order is the inheritance hierarchy, and in languages that support type conversion, the relation in the partial order is the conversion relation, such as int $\leq$ double, which means that an integer can be converted to a double. This latter relation is sometimes considered as subtyping between primitive data types [12]. In the Ptolemy II data type system, the type hierarchy if further constrained to be a lattice, and type constraints are formulated and solved over the lattice [15].

We form a polymorphic type system at the system-level through an approach similar to subtyping. Using the alternating simulation relation of interface automata, we organize all the interaction types in a partial order. Given this hierarchy, if a component is compatible with a certain type A, it is also compatible with all the subtypes of A. This property can be used to facilitate the design of polymorphic components and simplify type checking.

Even with the power of polymorphism, no type system can capture all the properties of programs and allow type checking to be performed efficiently while keeping the language flexible. So the language designer always has to decide what properties to

include in the system and what to leave out. Furthermore, some properties that can be captured by types cannot be easily checked statically before the program runs. This is either because the information available at compile time is not sufficient, or because that checking those properties is too costly. Hence, the designer also needs to decide whether to check those properties statically or at run time. Any type system represents some compromise. For example, array bound checking is vary helpful in detecting program errors, but it is hard to do efficiently by static checks. Some languages, such as C, do not perform this check. Other languages, such as ML and Java, perform the check, but at run time, and at the cost of run time performance. Some researchers propose to perform this check at compile time [14], but the technique requires the programer to insert annotations in the source code, since modern languages do not include array bounds in their type systems.

Type systems at the system-level have similar trade-offs. Among all the properties in a component-based design environment, we choose to check the compatibility of communication protocols as the starting point. This is because communication protocols are the central piece in many models of computation [8] and determine many other properties in the models. Our type system is extensible so other properties, such as deadlock in concurrent models, can be included in type checking. Another reason we choose to check the compatibility of communication protocols is that it can be done efficiently, when a component is inserted in a model. More complicated checking may need to be postponed to run time.

The rest of this paper is organized as follows. Section 2 describes Ptolemy II, with emphasis on the implementation of various communication protocols. Section 3 gives an overview of interface automata. Section 4 presents our system-level type system, including the type definition, the type hierarchy and some type checking examples. Section 5 discusses the trade-offs in the design of system-level types and run-time type checking. The last section concludes the paper and points out our future research directions.

2 Ptolemy II - A Component-Based Design Environment

Ptolemy II [3] is a system-level design environment that supports component-based heterogeneous modeling and design. The focus is on embedded systems. In Ptolemy II, components are called *actors*, and the channel of communication between actors is implemented by an object called a *receiver*, as shown in figure 1. Receivers are contained in *IOPorts* (input/output ports), which are in turn contained in actors.

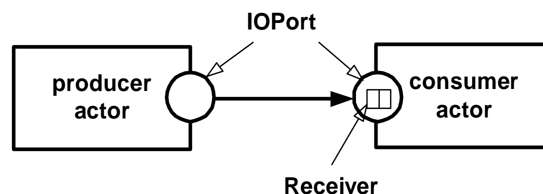


Fig. 1. A simple model in Ptolemy II.

Ptolemy II is implemented in Java. The methods in the receiver are defined in a Java interface Receiver. This interface assumes a producer/consumer model, and communicated data is encapsulated in a class called Token. The put() method is used by the producer to deposit a token into a receiver. The get() method is used by the consumer to extract a token from the receiver. The hasToken() method, which returns a boolean, indicates whether a call to get() will trigger a NoTokenException.

Aside from assuming a producer/consumer model, the Receiver interface makes no further assumptions. It does not, for example, determine whether communication between actors is synchronous or asynchronous. Nor does it determine the capacity of a receiver. These properties of a receiver are determined by concrete classes that implement the Receiver interface. Each one of these concrete classes is part of a Ptolemy II *domain*, which is a collection of classes implementing a particular model of computation. In each domain, the receiver determines the communication protocol, and an object called a *director* controls the execution of actors. From the point of view of an actor, the director and the receiver form its execution environment.

Each actor has a fire() method that the director uses to start the execution of the actor. During the execution, an actor may interact with the receivers to receive or send data. Some of the domains in Ptolemy II are:

- *Communicating Sequential Processes (CSP)*: As the name suggests, this domain implements a rendezvous-style communication (sometimes called synchronous message passing), as in Hoare's communicating sequential processes model [5]. In this domain, the producer and consumer are separate threads executing the fire() method of the actors. Which ever thread calls put() or get() first blocks until the other thread calls get() or put(). Data is exchanged in an atomic action when both the producer and consumer are ready.
- *Process Networks (PN)*: This domain implements the Kahn process networks model of computation [6]. The Ptolemy II implementation is similar to that by Kahn and MacQueen [7]. In that model, just like CSP, the producer and consumer are separate threads executing the fire() method. Unlike CSP, however, the producer can send data and proceed without waiting for the receiver to be ready to receive data. This is implemented by a non-blocking write to a FIFO queue with (conceptually) unbounded capacity. The put() method in a PN receiver always succeeds and always returns immediately. The get method, however, blocks the calling thread if no data is available. To maintain determinacy, it is important that processes not be able to test a receiver for the presence of data. So the hasToken() method always returns *true*. Indeed, this return value is correct, since the get() method will never throw a NoTokenException. Instead, it will block the calling thread until a token is available.
- *Synchronous Data Flow (SDF)*: This domain supports a synchronous dataflow model of computation [9]. This is different from the thread-based domains in that the producer and consumer are implemented as finite computations (firings of a dataflow actor) that are scheduled (typically statically, and typically in the same thread). In this model, a consumer assumes that data is always available when it calls get() because it assumes that it would not have been scheduled otherwise. The capacity of the receiver can be made finite, statically determined, but the

scheduler ensures that when put() is called, there is room for a token. Thus, if scheduling is done correctly, both get() and put() succeed immediately and return.
- *Discrete Event (DE)*: This domain uses timed events to communicate between actors. Similar to SDF, actors in the DE domain implement finite computations encapsulated in the fire() method. However, the execution order among the actors is not statically scheduled, but determined at run time. Also, when a consumer is fired, it cannot assume that data is available. Very often, when an actor with multiple input ports is fired, only one of the ports has data. Therefore, for an actor to work correctly in this domain, it must check the availability of a token using the hasToken() method before attempting to get a token from the receiver.

As can be seen, different domains impose different requirements for actors. Some actors, however, can work in multiple domains. These actors are called *domain-polymorphic* actors. One of the goals of the system-level type system is to facilitate the design of domain-polymorphic actors.

In Ptolemy II, there are more than ten domains implementing various models of computation, including the ones discussed above. One of these domains implements interface automata.

3 Overview of Interface Automata

3.1 An Example

Interface automata [4] are a light-weight formalism for the modeling of components and their environments. As other automata models, interface automata consist of states and transitions[1], and is usually depicted by bubble-and-arc diagrams. There are three different kinds of transitions in interface automata: input, output, and internal transitions. When modeling a software component, input transitions correspond to the invocation of methods on the component, or the returning of method calls from other components. Output transitions correspond to the invocation of methods on other components, or the returning of method calls from the component being modeled. Internal transitions correspond to computations inside the component.

For example, figure 2 shows the interface automata model for an implementation of the consumer actor in figure 1. This figure is a screen shot of Ptolemy II. The convention in interface automata is to label the input transitions with an ending “?”, the output transitions with an ending “!”, and internal transitions with an ending “;”. Figure 2 does not contain any internal transitions. The block arrows on the sides of figure 2 denote the inputs and outputs the automaton. They are:
- *f*: the invocation of the fire() method of the actor.
- *fR*: the return from the fire() method.
- *g*: the invocation of the get() method of the receiver at the input port of the actor.
- *t*: the token returned in the get() call.
- *hT*: the invocation of the hasToken() method of the receiver.
- *hTT*: the value *true* returned from the hasToken() call, meaning that the receiver

[1]Transitions are called actions in [4].

contains one of more tokens.

- *hTF*: the value *false* returned from the hasToken() call, meaning that the receiver does not contain any token.

The initial state is state 0. When the actor is in this state, and the fire() method is called, it calls get() on the receiver to obtain a token. After receiving the token in state 3, it performs some computation, and returns from fire(). This example illustrates an important characteristic of interface automata. That is, they do not require all the states to accept all inputs. In figure 2, the input *f* is only accepted in state 0, but not in any other states. This is opposed to other automata-based formalisms, such as I/O automata [11], where every input must be enabled at every state. By not requiring the model to be input enabled, interface automata models are usually more concise, and do not include states that model error conditions. In fact, interface automata take an optimistic approach for modeling, and they reflect the intended behavior of components under a good environment.

In the SDF domain, an actor assumes that its fire() method will not be called again if it is already inside this method. Also, the scheduler guarantees that data is available when a consumer is fired, so the transition from state 2 to state 3 assumes that the receiver will return a token. An error condition, such as the receiver throws NoToken-Exception when get() is called, is not explicitly described in the model.

3.2 Composition and Compatibility

Two interface automata can be composed if their transitions are disjoint, except that an input transition of one may coincide with an output transition of the other. These overlapping transitions are called shared transitions. Shared transitions are taken synchronously, and they become internal transitions in the composition. Figure 3 shows two automata that can be composed with the automaton in figure 2. These two automata do not correspond to real Ptolemy II components, we just use them to illustrate automata composition. The *DirectorA* in figure 3(a) calls the fire() method of the SDFActor, then expects the call to return. When composed with the *SDFActor* automaton, *f* and *fR*

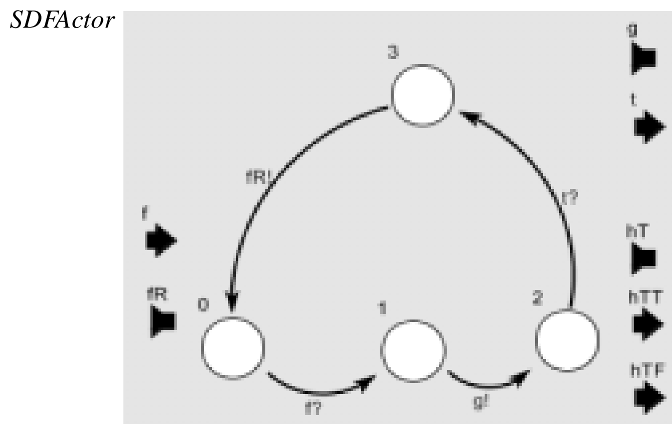
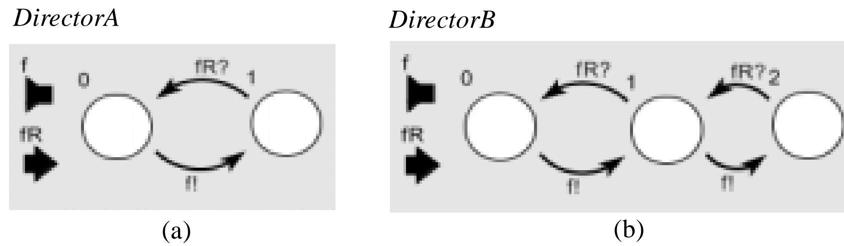


Fig. 2. Interface automaton model for an SDF actor.

are shared transitions, and the composition result is shown in figure 4(a). The *DirectorB* in figure 3(b) may call the fire() method again before the first call returns. When this automaton is composed with *SDFActor*, it may issue an output that the *SDFActor* does not accept. For example, when both automata are in state 1, *DirectorB* may issue *f*, which *SDFActor* does not accept. This means that the pair of states (1, 1) in the composition (*DirectorB*, *SDFActor*) is *illegal*.

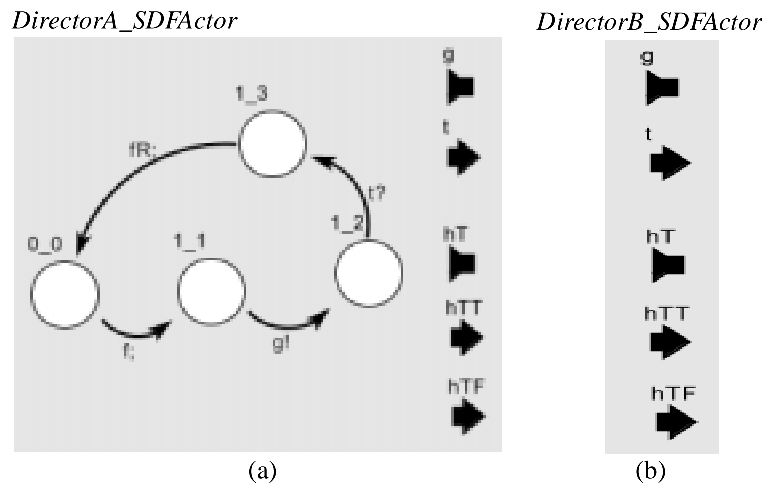
In interface automata, illegal states are pruned out in the composition. Furthermore, all states that can reach illegal states through output or internal transitions are also pruned out. This is because the environment cannot prevent the automata from entering illegal states from these states. As a result, the composition of *DirectorB* and *SDFActor* is an empty automaton without any states, as shown in figure 4(b). This is a key property of interface automata. More conventional automaton composition always results in a state space that is the product of the composed state spaces, and hence is significantly larger. Interface automata often compose to form smaller automata.

The above examples illustrate the key notion of *compatibility* in interface automata. Two automata are compatible if their composition is not empty. This notion gives a



(a) (b)

Fig. 3. Two (artificial) director automata.



(a) (b)

Fig. 4. The composition of the artificial directors (figure 3) with *SDFActor* (figure 2).

formal definition for the informal statement “two components can work together”. The composition automaton defines exactly how they can work together. In system-level types, we use interface automata to describe various communication protocols, or the interaction types for components. To check whether a certain component is compatible with a communication protocol, we can simple compose the automata models of the component and the protocol, and check whether the result is empty. This yields a straightforward algorithm for type checking, which is the main attraction of interface automata to system-level types.

The approach to composition in interface automata is optimistic. If two components are compatible, there is some environment that can make them work together. In the traditional pessimistic approach, two components are compatible if they can work together in all environments. Because of this difference, the composition of interface automata is usually smaller than the composition in other automata models. Before adopting interface automata, we also attempted to describe system-level types using a more traditional finite state machine model [10]. Compatibility checking in that setting proved to be more difficult.

3.3 Alternating Simulation

Interface automata have a notion of *alternating simulation*, which is used in our type system to define subtyping. Informally, for two interface automata P and Q, there is an alternating simulation relation from Q to P if all the input steps of P can be simulated by Q, and all the output steps of Q can be simulated by P. The formal definition is given in [4]. A theorem states that if a third automaton R is compatible with P, and the input transitions of Q that are shared with the output transitions of R is a subset of the input transitions of P that are shared with the output transitions of R, then Q and R are also compatible.

4 System-Level Types

4.1 Type Definition

The automaton *SDFActor* in figure 2 describes a consumer actor designed for the SDF (synchronous dataflow) domain. The automaton shown in figure 5 describes an actor that can operate in wider variety of domains. Since this actor is not designed under the assumption of the SDF domain, it does not assume that data are available when it is fired. Instead, it calls hasToken() on the receiver to check the availability of a token. If hasToken() returns false, it immediately returns from fire(). This is a simple form of domain-polymorphism.

In Ptolemy II, actors interact with the director and the receivers of a domain. In figures 2 and 5, the block arrows on the left side denote the interface with the director, and the ones on the right side denote the interface with the receiver. As discussed in section 2, the implementation of the director and the receiver determines the semantics of component interaction in a domain, including the flow of control and the communication protocol. If we use an interface automaton to model the combined behavior of the director and the receiver, this automaton is then the type signature for the domain. Figure 6 shows such an automaton for the SDF domain. Here, p and pR represent the

call and the return of the put() method of the receiver. This automaton encodes the assumption of the SDF domain that the consumer actor is fired only after a token is put into the receiver[1].

The type signature of the DE domain is shown in figure 7. In DE, an actor may be fired without a token being put into the receiver at its input. This is indicated by the transition from state 0 to state 7. Figures 6 and 7 also reflect the fact that both of the

PolyActor

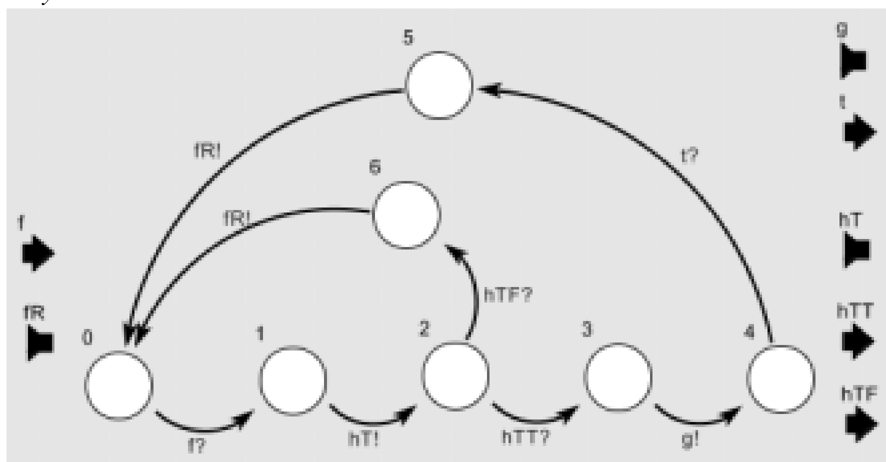


Fig. 5. Interface automaton for a domain-polymorphic actor.

SDFDomain

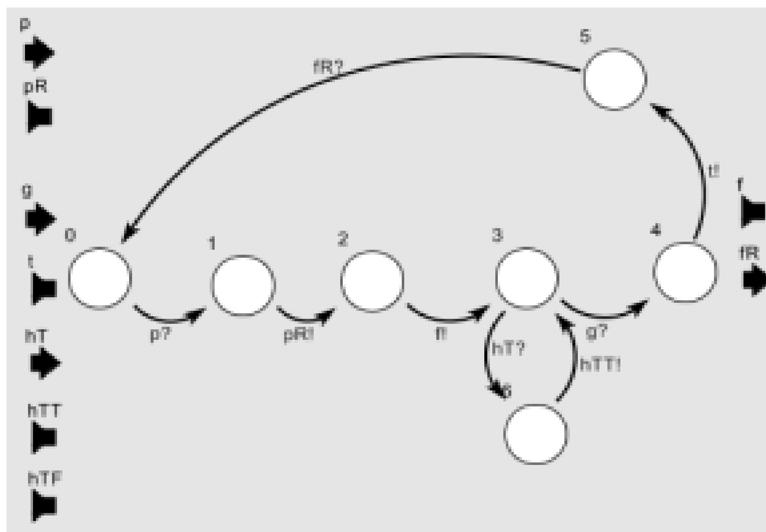


Fig. 6. Type signature of the SDF domain.

SDF and the DE domains have a single thread of execution, so the hasToken() query may happen only after the actor is fired but before it calls get(), during which time the actor has the thread of control.

CSP and PN are two domains in Ptolemy II in which each actor runs in its own thread. Figures 8 and 9 give the type signature of these two domains. These automata are simplified from the true implementation in Ptolemy II. In particular, CSPDomain omits conditional rendezvous, which is an important feature in the CSP model of computation. In the CSP and PN domains, an actor is fired repeatedly by its thread, as modeled by the transitions between state 0 and 1.

In CSP, the communication is synchronous; the first thread that calls get() or put() on the receiver will be stalled until the other thread calls put() or get(). The case where get() is called before put() is modeled by the transitions among the states 1, 3, 4, 5, 1. The case where put() is called before get() is modeled by the transitions among the states 1, 6, 8, 9, 1.

In PN, the communication is asynchronous. So the put() call always returns immediately, but the thread calling get() may be stalled until put() is called. The case where get() is called first in PN is modeled by the transitions among the states 1, 3, 4, 5, 1 in figure 9, while the case where put() is called first is modeled by the transitions among the states 1, 6, 8, 10, 1.

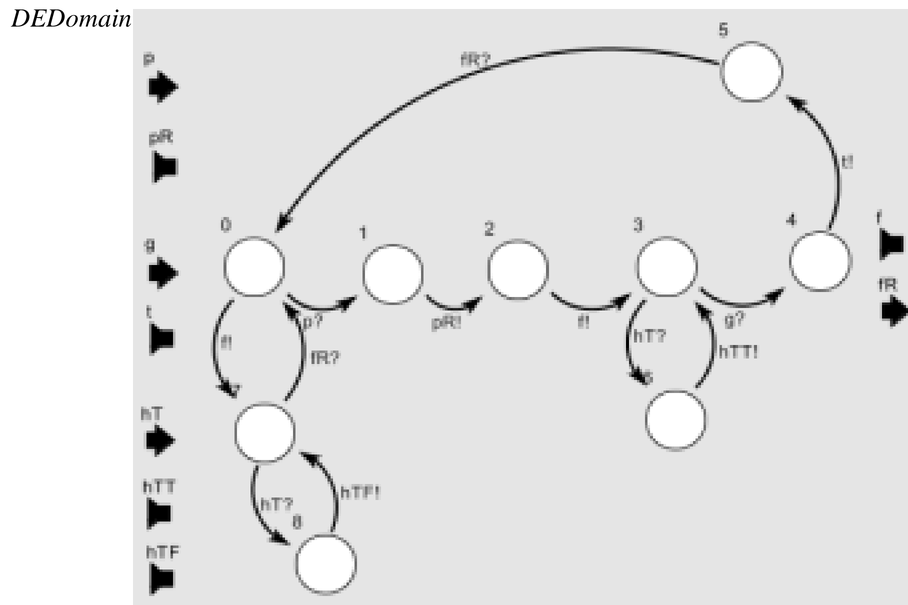


Fig. 7. Type signature of the DE domain.

---

[1] This is a simplification of the SDF domain, since an actor may require more than one token to be put in the receiver before it is fired. This simplification makes our exposition clearer, but otherwise makes no material difference.

Given an automaton modeling an actor and the type signature of a domain, we can check the compatibility of the actor with the communication protocol of that domain by composing these two automata. Type checking examples will be shown below in section 4.3.

SPDomain

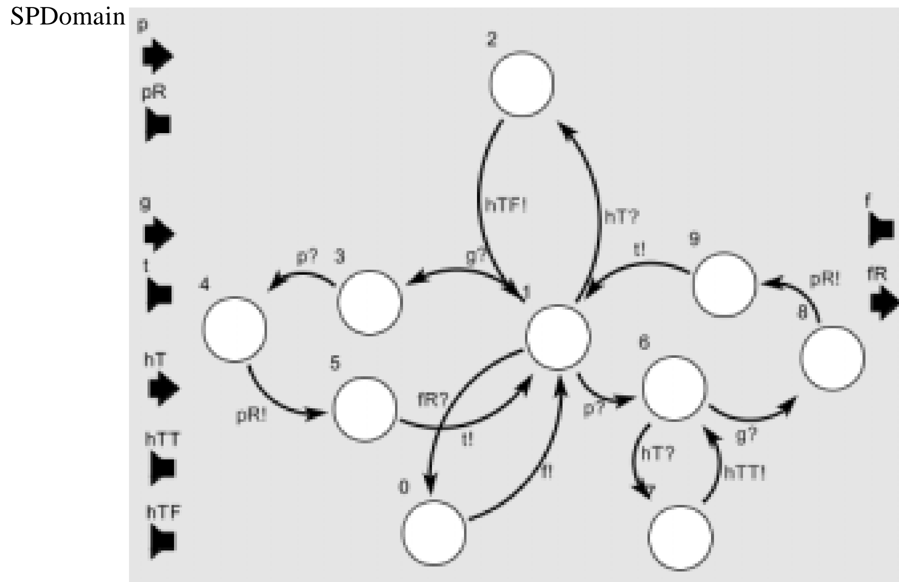


Fig. 8. Type signature of the CSP domain.

PNDomain

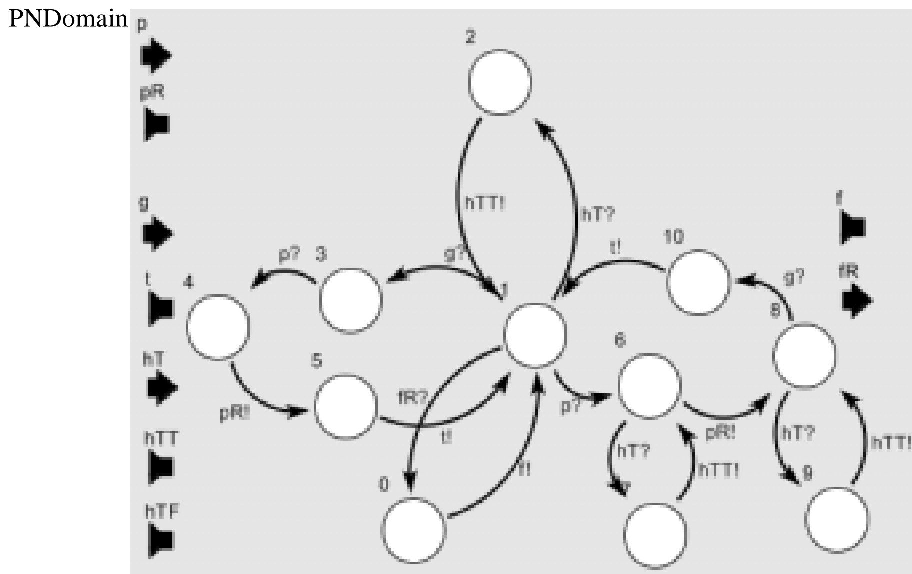


Fig. 9. Type signature of the PN domain.

4.2 System-Level Type Order and Polymorphism

If we compare the SDF and DE domain automata, we can see that they are closely related. This relationship can be captured by the alternating simulation relation of interface automata. In particular, there is an alternating simulation relation from SDF to DE.

In the set of all domain types, the alternating simulation relation induces a partial order, or system-level type order. An example of this partial order is shown in figure 10. From a type system point of view, this order is the subtyping hierarchy for the domain types. If we view the automata as functions with inputs and outputs, then the alternating simulation relation is exactly analogous to the standard function subtyping relation in data type systems. According to the definition of alternating simulation, the automaton lower in the hierarchy can simulate all the input steps of the ones above it, and the automaton higher in the hierarchy can simulate all the output steps below it. In function subtyping, if a function $A \rightarrow B$ is a subtype of another function $A' \rightarrow B'$, then A' is a subtype of A and B is a subtype of B' [1]. Note that in both relations, the order is inverted (contravariant) for the inputs and goes in the same direction (covariant) for the outputs.

In [4], alternating simulation is used to capture the refinement relation from the specification to the implementation of components. Our use of this relation is not for refinement, but for subtyping. In the system-level type order, *SDFDomain* is not a refinement of *DEDomain*, but a subtype of *DEDomain*. In fact, *SDFDomain* has fewer states than *DEDomain*. This subtyping relation can help us design actors that can work in multiple domains. According to the theorem in section 3, if an actor is compatible with a certain domain *D*, and there are other domains below D in the system-level type order, then the actor is also compatible with those lower domains. Therefore, this actor is domain polymorphic.

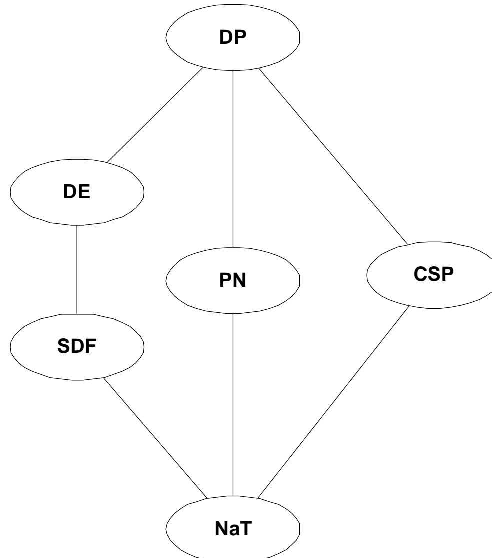


Fig. 10. An example of system-level type order.

In figure 10, we added a bottom and a top element to the type hierarchy. The name of the bottom element *NaT* stands for “not a type”, and the name of the top element *DP* stands for “domain polymorphic”. The DP automaton has an alternating simulation relation from all the domain-specific automata. So if an actor is compatible with this automaton, it is compatible with all the domains. The exact design of this automaton is part of our future research, and depends on the set of domains to be included.

Note that by adding the top and bottom elements, the system-level type order becomes a lattice. In a lattice, every subset of elements has a least upper bound and a greatest lower bound. It might be possible to explore this fact in the type system, as we have done for data types [15].

4.3 Type Checking Examples

Let’s perform a few type checking operations using the actors and domains in the earlier sections. To verify that the *SDFActor* in figure 2 can indeed work in the *SDFDomain*, we compose it with the *SDFDomain* automaton in figure 6. The result is shown in figure 11. As expected, the composition is not empty so *SDFActor* is compatible with *SDFDomain*.

Now let’s compose *DEDomain* with *SDFActor*. The result is an empty automaton shown in figure 12. This is because the actor may call get() when there is no token in the receiver, and this call is not accepted by an empty DE receiver. The exact sequence

SDFDomain_SDFActor

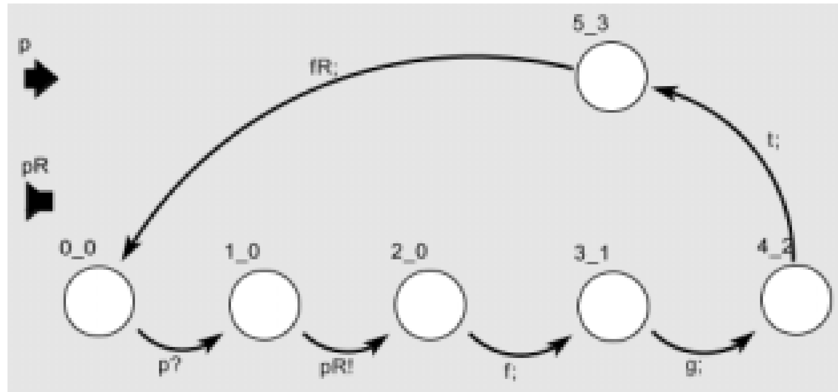


Fig. 11. Composition of *SDFDomain* in figure 6 and *SDFActor* in figure 2.

DEDomain_SDFActor

Fig. 12. Composition of *DEDomain* in figure 7 and *SDFActor* in figure 2.

that leads to this condition is the following: first, both automata take a shared transition *f*. In this transition, *DEDomain* moves from state 0 to state 7, and *SDFActor* moves from state 0 to state 1. At state 1, *SDFActor* issues *g*, but this input is not accepted by *DEDomain* at state 7. So the pair of states (7, 1) in (*DEDomain*, *SDFActor*) is illegal. Since this state can be reached from the initial state (0, 0), the initial state is pruned out from the composition. As a result, the whole composition is empty. This means that the SDF actor cannot be used in DE Domain.

The *PolyActor* in figure 5 checks the availability of a token before attempting to read from the receiver. By composing it with *DEDomain*, we verify that this actor can be used in the DE Domain. This composition is shown in figure 13. Since *SDFDomain* is below *DEDomain* in the system-level type order of figure 10, we have also verified that *PolyActor* can work in the SDF domain. Therefore, *PolyActor* is domain polymorphic. As a sanity check, we have composed *SDFDomain* with *PolyActor*, with the result is shown in figure 14.

We have also composed *PolyActor* and *SDFActor* with *CSPDomain* and *PNDomain*. The result shows that these two actors can be used in both domains. This is not surprising, because both domains have blocking read semantics, so the actors will work regardless of whether they check the availability of a token before calling get(). For the sake of brevity, we do not include these compositions in this paper.

In Ptolemy II, there is a library of about 100 domain-polymorphic actors. The communication behavior for many of these actors can be modeled by the *PolyActor* automaton.

DEDomain_PolyActor

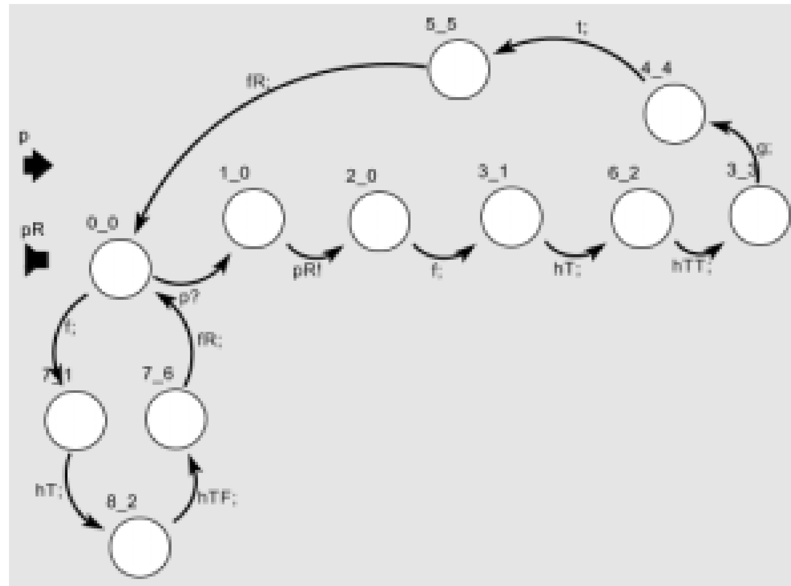


Fig. 13. Composition of *DEDomain* in figure 7 and *PolyActor* in figure 5.

4.1 Reflection

So far, interface automata have been used to describe the operation of Ptolemy II components. These automata can be used to perform compatibility checks between components. Another interesting use is to reflect the component state in a run-time environment. For example, we can execute the automaton *SDFActor* of figure 2 in parallel with the execution of the actor. When the fire() method of the actor is called, the automaton makes a transition from state 0 to state 1. At any time, the state of the actor can be obtained by querying the state of the automaton. Here, the role of the automaton is reflection, as realized for example in Java. In Java, the Class class can be used to obtain the static structure of an object, while our automata reflect the dynamic behavior of a component. We call an automaton used in this role a *reflection automaton.*

5 Trade-offs in Type System Design

The examples in the previous section focus on the communication protocol between a single actor and its environment. This scope can be broadened by including the automata of more actors and using a more detailed director model in the composition. Also, properties other than the communication protocol, such as deadlock freedom in thread-based domains, can be included in the type system. However, these extensions will increase the cost of type checking. So there is a trade-off between the amount of information carried by the type system and the cost of type checking.

Another dimension of the trade-offs is static versus run-time type checking. The examples in the last section are static type checking examples. If we extend the scope of the type system, static checking can quickly become impractical due to the size of the composition. An alternative is to check some of the properties at run time. One way to perform run-time checking is to execute the reflection automata of the components in parallel with the execution of the components. Along the way, we periodically check the states of the reflection automata, and see if something has gone wrong.

These trade-offs imply that there is a big design space for system-level types. In this space, one extreme point is completely static checking by composing the automata modeling all the system components, and check the composition. This amounts to model checking. To explore the boundary in this direction, we did an experiment by

SDFDomain_PolyActor

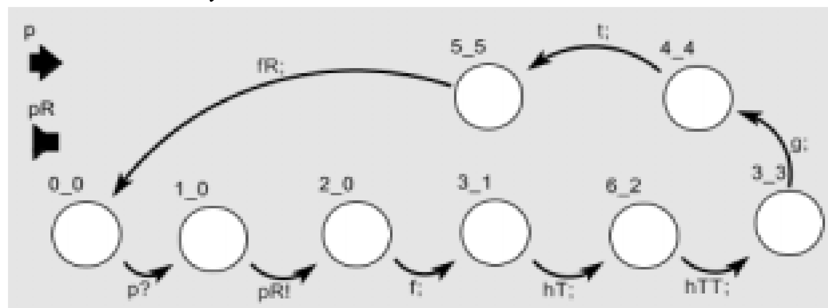


Fig. 14. Composition of *SDFDomain* in figure 6 and *PolyActor* in figure 5.

checking an implementation of the classical dining philosophers model implemented in the CSP domain in Ptolemy II. Each philosopher and each chopstick is modeled by an actor running in its own thread. The chopstick actor uses conditional send to simultaneously check which philosopher (the one on its left or the one on its right) wants to pick it up. We created interface automata for the Ptolemy II components *CSPReceiver*, *Philosopher*, and *Chopstick*, and a simplified automaton to model conditional send. We are able to compute the composition of all the components in a two-philosopher version of the dining philosopher model, and obtain a closed automaton with 2992 states. Since this automaton is not empty, we have verified that the components in the composition are compatible with respect to the synchronous communication protocol in CSP. We also checked for deadlock inherent in the implementation, and are able to identify two deadlock states in the composition, which correspond to the situation where all the philosophers are holding the chopsticks on their left and waiting for the ones on the right, and the symmetrical situation where all philosophers are waiting for the chopsticks on their left.

Our goal here is not to do model checking, but to perform static type checking on a non-trivial models. Obviously, when the model grows, complete static checking will become intractable due to the well-known state explosion problem.

Another extreme point in the design space for system-level types is to rely on run-time type checking completely. For deadlock detection, we can execute the reflection automata in parallel with the Ptolemy II model. When the model deadlocks, the states of the automata will explain the reason for the deadlock. In this case, the type system becomes a debugging tool. The point here is that a good type system is somewhere between these extremes. We believe that a system that checks the compatibility of communication protocols, as illustrated in sections 4, is a good starting point.

6 Conclusion and Future Work

We have described a type system that captures the interaction dynamic in a component-based design environment. The interaction types and component behavior are described by interface automata, and type checking is done through automata composition. Our approach is domain polymorphic in that a component may be compatible with multiple interaction types. The relation among the interaction types is captured by a system-level type order using the alternating simulation relation of interface automata. We have shown that our system can be extended to capture more dynamic properties, and the design of a good type system involves a set of trade-offs. Our experimental platform is the Ptolemy II design environment. All the automata in the paper are built in Ptolemy II and their compositions are computed in software, except that some manual layout is applied for better readability of the diagrams.

We also proposed using automata to do on-line reflection of component states. In addition to run-time type checking, the resulting reflection automata can add value in a number of ways. For example, in a reconfigurable architecture or distributed system, the state of the reflection automata can provide information on when it is safe to perform mutation. Reflection automata can also be valuable debugging tools. This is part of our future work.

In addition to its usual use in type checking, our type system may facilitate the design of new components or Ptolemy II domains. In Ptolemy II, domains can be combined hierarchically in a single model. Using system-level types, it might be possible to show that the composition of a domain director and a group of actors behaves like a polymorphic actor in some other domains. This is also part of our future research.

Acknowledgments

We thank Xiaojun Liu for his help in the implementation of interface automata in Ptolemy II. This work is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: Agilent, Cadence, Hitachi, and Philips.

References

1. L. Cardelli and P. Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,” *ACM Computing Surveys*, Vol.17, No.4, Dec. 1985.

2. B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

3. J. Davis II, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong, “Heterogeneous Concurrent Modeling and Design in Java,” *Technical Memorandum UCB/ERL M01/12*, EECS, University of California, Berkeley, March 15, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/HMAD/)

4. L. de Alfaro and T. A. Henzinger, “Interface Automata,” to appear in *Proc. of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 01)*, Austria, 2001.

5. C. A. R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, 28(8), August 1978.

6. G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

7. G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

8. E. A. Lee, “Computing for Embedded Systems,” *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.

9. E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *Proc. of the IEEE*, Sept., 1987.

10. E. A. Lee and Yuhong Xiong, “System-Level Types for Component-Based Design,” *Technical Memorandum UCB/ERL M00/8*, EECS, University of Cali-

fornia, Berkeley, Feb. 29, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/systemLevel/)

11. N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th ACM Symp. Principles of Distributed Computing*, pp 137-151, 1981.

12. J. C. Mitchell, "Coercion and Type Inference," *11th Annual ACM Symposium on Principles of Programming Languages*, 175-185, 1984.

13. M. Odersky, "Challenges in Type System Research," *ACM Computing Surveys*, 28(4), 1996.

14. H. Xi and F. Pfenning, "Eliminating Array Bound Checking Through Dependent Types," *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '98)*, pp. 249-257, Montreal, June, 1998.

15. Y. Xiong and E. A. Lee, "An Extensible Type System for Component-Based Design," *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. LNCS 1785.