Behavioral 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, including the communication protocols between components. Our system is based on a light-weight formalism - interface automata. We first propose some extensions to the theory of interface automata to make them more powerful, then use the extended theory to establish a behavioral-level type system. In our system, the interaction types and the dynamic behavior of components are defined using interface automata, and 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 that this relation can be described using a partial order. This behavioral 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 behavioral type systems.

Keywords: Behavioral type; Component-based design; Interface automata; Polymorphism; Alternating simulation

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 environments. Fundamentally, a type system detects mismatches 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 [XiL00] 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 [Lee02] for many more examples). So far, most type system research for component-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 *behavioral types*. 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* [deH01] 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 [BCD02].

We have found that the design of behavioral 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 [Ode96]. As mentioned earlier, modern polymorphic type systems have achieved this goal to a large extend. At the behavioral 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 [CaW85], 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 [DaP90]. 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 [Mit84]. 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 [XiL00][Xio02].

We form a polymorphic type system at the behavioral 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 very 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 [XiP98], 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 behavioral 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 [Lee02] 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.

In our earlier work [LeX01], we use interface automata to specify the interaction types and use alternating simulation as the subtyping relation. Recently, we observed that the original interface automata model needs some extensions to work better in more situations, and some of the relations among behavioral types are not directly captured by alternating simulation. We propose to extend the theory of interface automata to address these issue, and report these extensions in this paper.

The rest of this paper is organized as follows. Section 2 gives an overview of interface automata. Section 3 discusses our extensions. Section 4 describes Ptolemy II, with emphasis on the implementation of various communication protocols. Section 5 presents our behavioral type system, including the type definition, the type hierarchy and some type checking examples. Section 6 discusses some issues in the behavioral type systems and related works. The last section concludes the paper and points out our future research directions.

2. Overview of Interface Automata

2.1 An Example

Interface automata [deH01] are a light-weight formalism for the modeling of components and their environments. As with other automata models, interface automata consist of states and transitions[1], and are 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.

In behavioral-level modeling, one of the most frequently used examples is buffered communication. We will use interface automata to model such a scenario. Assume we have two software components, a *Producer* and a *Consumer*, and that they communicate through a one-place buffer. The buffer component has the following methods: put(), get(), hasRoom(), and hasToken(). The producer uses hasRoom() to check whether the buffer has room for a token. If this method returns *true*, it calls the put() method to put a token into the buffer. Similarly, the consumer uses hasToken() to check whether the buffer contains a token. If this method returns *true*, it calls get() to extract the token. For the moment, let's just model the part of the buffer interface used by the consumer. We will add the interface for producer in later examples. Figure 1 shows the interface

[1]Transitions are called actions in [deH01].

automata model for the buffer. This and the subsequent figures are drawn in the Ptolemy II software [BCD02]. 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 ";". The block arrows on the sides of figure 1 denote the inputs and outputs of the automaton. They are:

- *g*: the invocation of the get() method of the buffer.
- *t*: the token returned in the get() call.
- *hT*: the invocation of the hasToken() method of the buffer.
- *hTT*: the value *true* returned from the hasToken() call, meaning that the buffer contains a token.
- *hTF*: the value *false* returned from the hasToken() call, meaning that the buffer does not contain a token.

Notice that the interaction with the producer is abstracted into one internal transition *p_pR*. Here, *p* denotes the invocation of the put() method, and *pR* denotes the return of the put() call. The initial state is state 0. When the actor is in this state, and the consumer queries whether there is a token by calling hasToken(), the receiver returns *false*. This call and its return is modeled by the transition from state 0 to 4, and 4 to 0. If the producer deposits a token, the automaton will move to state 2. At this state, the hasToken() call will return *true*. If the consumer calls get() at state 2, the buffer will return a token for this call. This is modeled by the transition from state 2 to 3, and 3 to 0.

This example illustrates an important characteristic of interface automata. That is, they do not require all the states to accept all inputs. In figure 1, the input *g* is only accepted at state 2, but not in any other states. This is opposed to other automata-based formalisms, such as I/O automata [LyT81], 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 to modeling, and they reflect the intended behavior of components under a good environment. Under this philosophy, error conditions are usually not explicitly modeled. For example, in figure 1, we do not have states and transitions to describe the case when get() is called on an empty buffer.

2.2 Composition and Compatibility

Two interface automata can be composed if the names of their transitions (excluding the "?, !, ;") 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 2 shows two con-

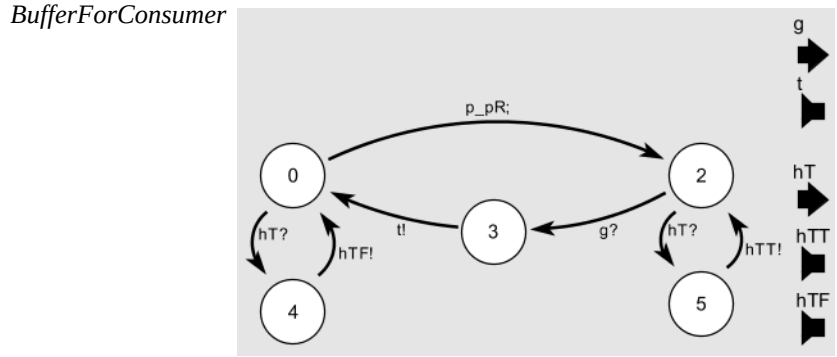
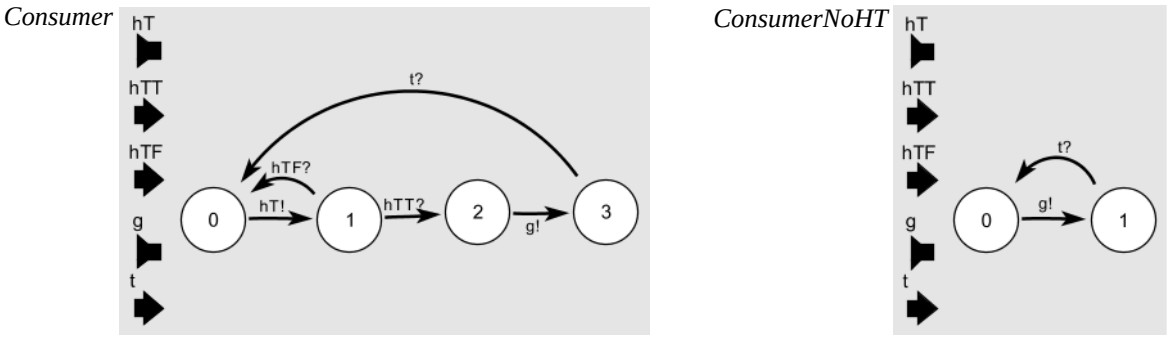


Figure 1. Interface automata model for a one-place buffer (only the consumer interface is modeled).

sumer automata that can be composed with the automaton *BufferForConsumer* in figure 1. The *Consumer* automaton in figure 2(a) keeps calling the hasToken() method of the buffer until it returns *true*, then calls the get() method to extract the token. When composed with the *BufferForConsumer* automaton, all the transitions are shared transitions, and the composition result is shown in figure 3(a). In figure 2(b), the consumer calls get() without first checking whether a token is available. When this automaton is composed with the buffer, it may issue an output that the buffer does not accept. For example, when both automata are in state 0, *ConsumerNoHT* may issue *g*, which *BufferForConsumer* does not accept. This means that the pair of states (0, 0) in the product automaton *BufferForConsumer*⊗*ConsumerNoHT* is *illegal*.

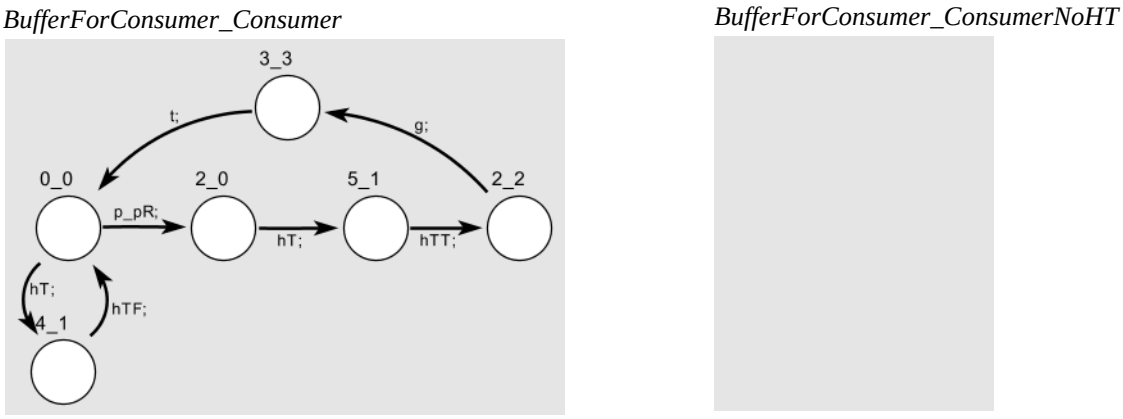
In interface automata, illegal states are pruned out in the composition. Furthermore, all states that can reach illegal states through output or internal transitions are also pruned out. This is because the environment cannot prevent the automata from entering illegal states from these states. As a result, the composition of *BufferForConsumer* and *ConsumerNoHT* is an empty automaton without any states, as shown in figure 3(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 formal definition for the informal statement “two components can work together”. The composition automaton



(a) (b)

Figure 2. Two consumer automata.



(a) (b)

Figure 3. Composition of *BufferForConsumer* in figure 1 and the two consumer automata in figure 2.

defines exactly how they can work together. In behavioral 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 simply 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 behavioral 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.

2.3 Alternating Simulation

Interface automata have a notion of *alternating simulation*, which is used to simplify type checking in our system. 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*. For example, the *BufferWithDefault* automaton in figure 4 models a buffer that can return a default token when it is empty. This automaton has an additional state, 6, compared to the one in figure 1, and the transition between state 0 and this state models the get() call and the return of the default token when the buffer is empty. There is an alternating simulation relation from *BufferWithDefault* to *BufferForConsumer*.

If there is an alternating simulation relation from *Q* to *P*, 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. In our example, since the *Consumer* automaton is compatible with *BufferForConsumer*, it is also compatible with *BufferWithDefault*.

3. Extensions to Interface Automata

3.1 Transient States

Suppose we want to extend the buffer example above to include the producer in the model. We can design a buffer like the one in figure 5, and a producer like the one in figure 6. In both figures, *p* and *pR* represent the call to the put() method and its return, and *hR*, *hRT*, *hRF* represent the call to hasRoom() and the two possible return values, *true* and *false*. These are the interaction between

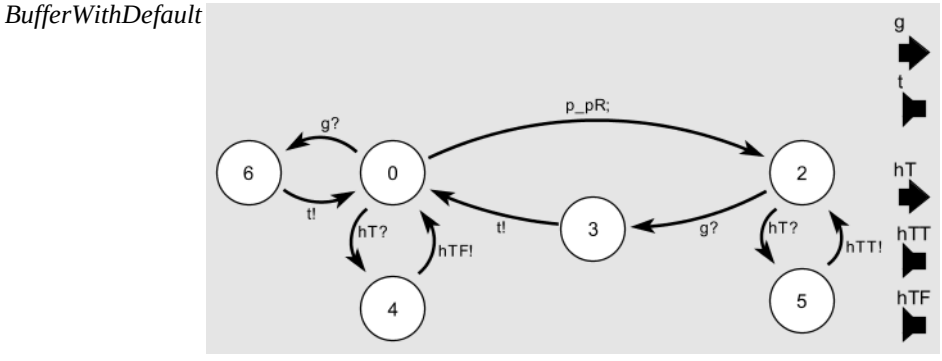


Figure 4. A buffer that can return a default token.

the buffer and the producer. Now, if we compose the *Producer* in figure 6, the *Buffer* in figure 5, and the *Consumer* in figure 2(a), the result is an empty automaton!

This result may be surprising, but if we examine these automata carefully, we can find many ways that the composition gets into illegal states. For example, if the producer calls hasRoom(), the *Producer* automaton moves to state 1, and the *Buffer* moves to state 6. At this time, if the consumer calls hasToken(), the *Buffer* automaton cannot accept this call at state 6, so the state (1, 6, 0) in *Producer*⊗*Buffer*⊗*Consumer* is illegal. Another situation where we enter illegal state is that the producer calls put(), but before this call is returned, the consumer calls hasToken(). Since these illegal states are reachable from the initial states of the automata, the whole composition is empty.

The issue here is that when we design the buffer like figure 5, we assume that its methods are non-interruptible. In fact, when we implement such a buffer in software, we probably will protect all of its methods, put(), get(), hasRoom(), and hasToken(), as critical sections. For example, if we implement these methods in Java, we will make them synchronized methods to achieve mutual exclusion. However, in interface automaton, there is an intermediate state between the input transition that represents a method call, and the output transition that represents the return of the call. So in the interface automaton model, the methods become interruptible.

If want to model the synchronization mechanism in Java, we will need to add an output "lock" and another "unlock," and any correct composition would have to check so that it never tries to send an automaton an input if it is locked. This would be really cumbersome. Since such mutexes are so common, we want to support them in the automata formalism. Moreover, they are more

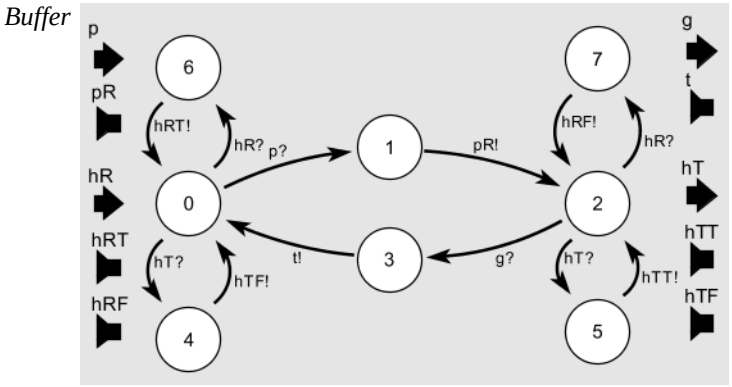


Figure 5. A one-place buffer.

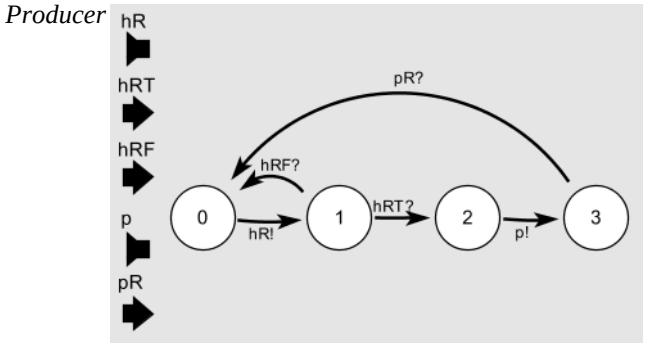


Figure 6. A producer.

common in interface automata than in ordinary automata because of the separation of inputs and outputs.

To do this, we introduce a notion of *transient* state. These are the above intermediate states. We denote these states with a "t" at the end of their names in our block diagrams, as shown in figure 7. Transient states can only have output and internal transitions. When we compose two automata, and one of the automata is in transient state, we do not take any output transition from the non-transient state, and just move along the output or internal transitions of the transient state. That is, the machine that is in a non-transient state stutters (remains in the same state and produces no output). We currently do not allow the composition to enter a pair of state where both of them are transient. The composition of the *Producer* in figure 6, the *BufferWithTransient* in figure 7, and the *Consumer* in figure 2(a) is shown in figure 8.

Thus, our model accurately reflects the behavior of critical sections. Notice that transient state is not required for traditional finite state machine (FSM) models [LeV01]. In FSM, we can combine an input and an output into one transition. For example, the FSM model for the buffer is

BufferWithTransient

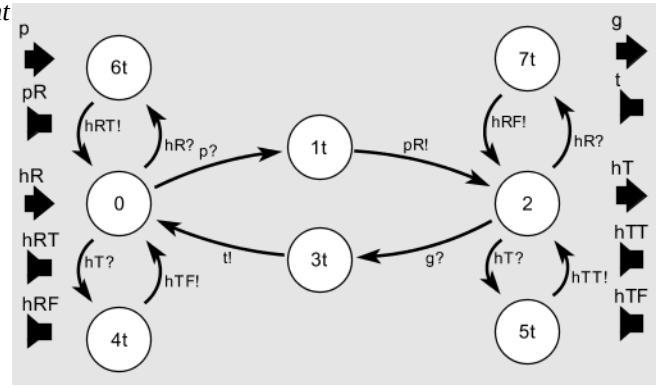


Figure 7. A buffer with transient states.

Producer_BufferWithTransient_Consumer

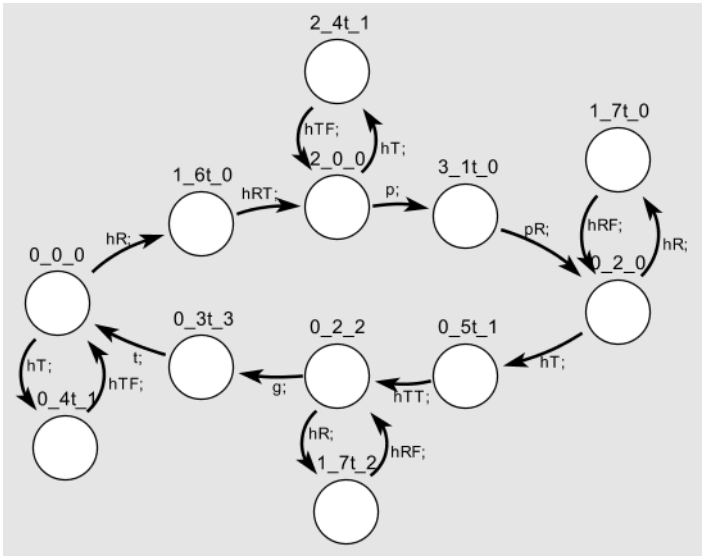


Figure 8. Composition of the *Producer* in figure 6, *BufferWithTransient* in figure 7, and *Consumer* in figure 2(a).

shown in figure 9. By adding the notion of transient states to interface automata, we achieve the ability in FSM to model non-interruptible input and output.

3.2 Projection Automata

In *BufferWithTransient*, there are four methods: put(), get(), hasRoom(), and hasToken(). Since the last two are not directly used for communication, they can be viewed as "overhead" of the communication. Suppose we want to study the amount of this overhead, we can count the number of times these two methods are called. To do this, we can update *BufferWithTransient* by sending out a "count" output every time hasRoom() or hasToken() is called. This buffer is shown in figure 10. Here, the output *c* represents the count event. It goes to a certain counter component. The design of the counter is not important here, so we omit its details.

Now, suppose we want to study the relation between the *BufferWithTransient* in figure 7 and the *BufferWithCounter* in figure 10, with respect to their compatibility with a consumer component. Intuitively, *BufferWithCounter* is a more refined model for the *BufferWithTransient*, so if a consumer can work with *BufferWithTransient*, it should work with *BufferWithCounter*. We would like to capture this using a formal relation between these two automata, such as alternating simulation. However, these two automata do not have an alternating simulation between them. If we analyze them carefully, we realize that the additional transition *c* in *BufferWithCounter* is "interfering" the alternating simulation. Even though this transition does not affect the consumer, it obscures the relation between the two buffer automata with respect to the consumer. To remove

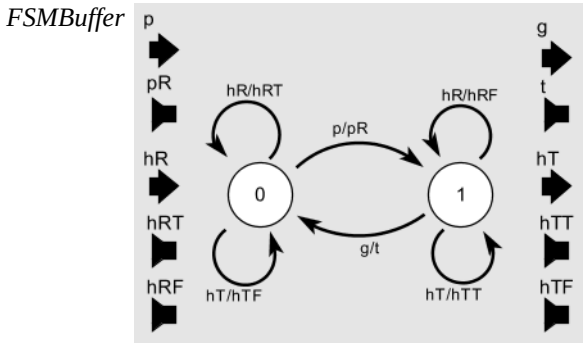


Figure 9. FSM model for the buffer in figure 7.

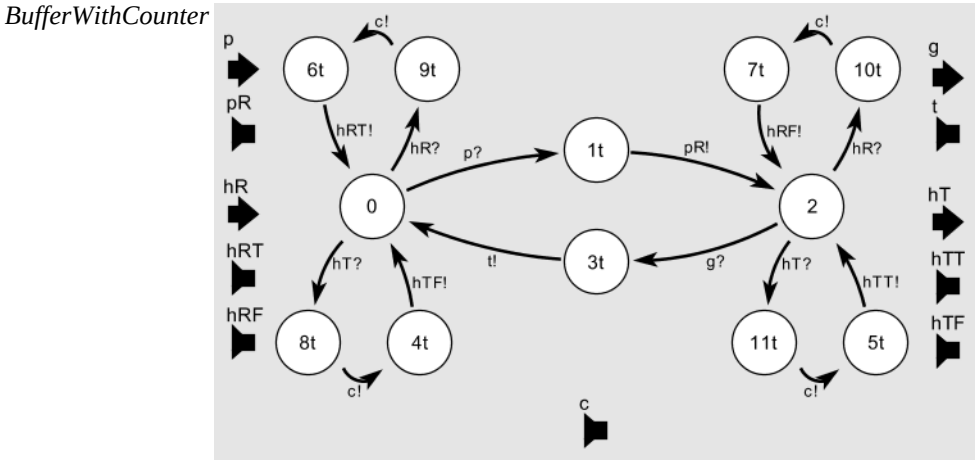


Figure 10. A buffer that counts the overhead methods.

this interference and restore the alternating simulation, we propose to "hide" these transitions from the external interface of the buffers. One way to do this hiding is to convert them to internal transitions. We can view this operation as a *projection* of the whole interface of the buffers to the subset of shared interface with *Consumer*. The projection of *BufferWithTransient* and *BufferWithCounter* to the *Consumer* are shown in figure 11 and figure 12, respectively.

Now, if we compute the alternating simulation between *BufferWithTransientToConsumer* and *BufferWithCounterToConsumer*, there is indeed an alternating simulation from *BufferWithCounterToConsumer* to *BufferWithTransientToConsumer*. So we have revealed the intuitive refinement relation between these two automata.

If an automaton is compatible with a projection automaton, it is also compatible with the original automaton. More specifically, if *P'* is the projection of *P* onto *R*, and *P'* and *R* are compatible, *P* and *R* are compatible. To see this, notice that the product automata $P \otimes R$ and $P' \otimes R$ have the same set of states and transitions, except that some of the transition labels are different. In particular, some of the input and output transitions in $P \otimes R$ are changed to internal transitions in $P' \otimes R$. Also, These two product automata have the same set of illegal states. Furthermore, for all the states in $P \otimes R$ that can reach illegal states through internal and output transitions, there corresponding states in $P' \otimes R$ can also reach the corresponding illegal states. In another word, when we prune out all the illegal states and all the states that can reach the illegal states through internal and output transitions in the two product automata, the set of states being pruned in $P' \otimes R$ is a super set of that of $P \otimes R$. So if the composition of $P' \otimes R$ is not empty, $P \otimes R$ is not empty.

BufferWithTransientToConsumer

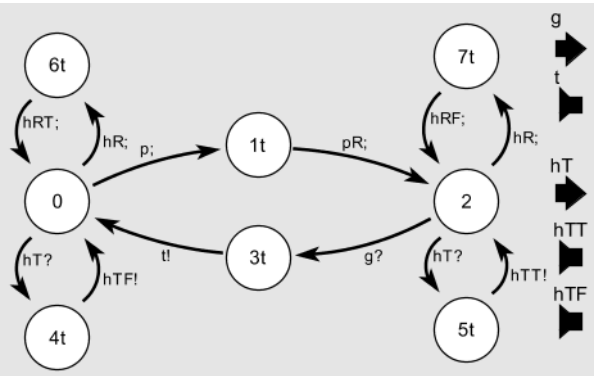


Figure 11. The projection of *BufferWithTransient* onto *Consumer*.

BufferWithCounterToConsumer

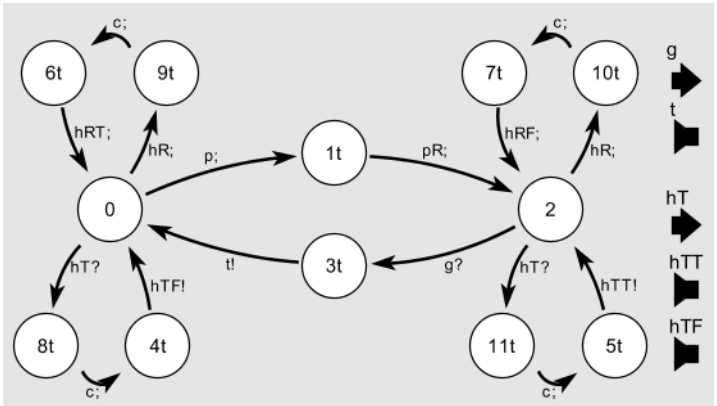


Figure 12. The projection of *BufferWithCounter* onto *Consumer*.

Given this, we have the following:

Given three interface automata *P*, *Q*, and *R*, let *P'* and *Q'* be the projections of *P* and *Q* onto *R*. If *P'* is compatible with *R*, and there is an alternating simulation from *Q'* to *P'*, then *Q* is compatible with *R*.

In our example, *P* is *BufferWithTransient, P'* is *BufferWithTransientToConsumer*, *Q* is *BufferWithCounter, Q'* is *BufferWithCounterToConsumer*, and *R* is *Consumer*. Since *Consumer* is compatible with *BufferWithTransientToConsumer*, it is also compatible with *BufferWithCounter*.

We can obtain similar result for the *Producer* automaton by symmetry.

The *BufferWithTransient*, *Producer*, and *Consumer* automata discussed in this model can be viewed as a particular implementation of a model of computation. In this model, the communication between the producer and the consumer is asynchronous, and their execution is not statically scheduled. There are many other models of computation with various nice properties, such as static schedulability and determinacy [Lee02]. Some of these models are implemented in the Ptolemy II environment.

4. Ptolemy II - A Component-Based Design Environment

Ptolemy II [BCD02] 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 13. Receivers are contained in *IOPorts* (input/output ports), which are in turn contained in actors.

Ptolemy II is implemented in Java. The methods in the receiver are defined in a Java interface called 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:

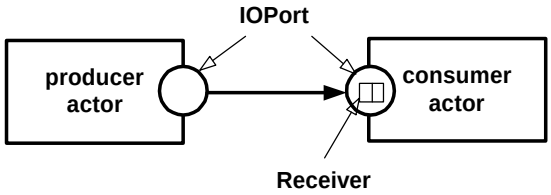


Figure 13. A simple model in Ptolemy II.

- *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 [Hoa78]. In this domain, the producer and consumer are separate threads executing the fire() method of the actors. Whichever 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 [Kah74]. The Ptolemy II implementation is similar to that by Kahn and MacQueen [KaM77]. 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 [LeM87]. 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 behavioral 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.

5. Behavioral Types

5.1 Type Definition

As we mentioned before, we use interface automata to describe the behavior of Ptolemy II components. In figure 14, the automaton *SDFConsumer* describes a consumer actor designed for the SDF (synchronous dataflow) domain. The inputs and outputs of the automaton are:

- *fC*: the invocation of the fire() method of the consumer actor.
- *fCR*: 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 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().

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 NoTokenException when get() is called, is not explicitly described in the model.

The automaton shown in figure 15 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.

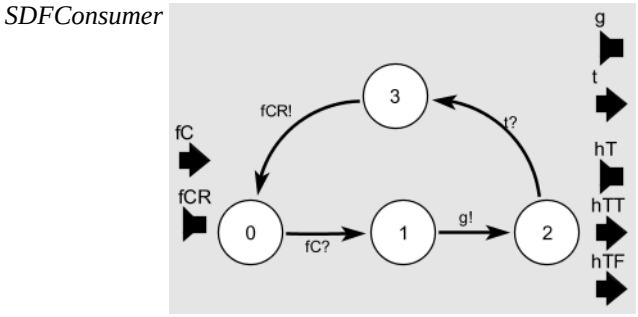


Figure 14. Interface automata model for an SDF consumer actor.

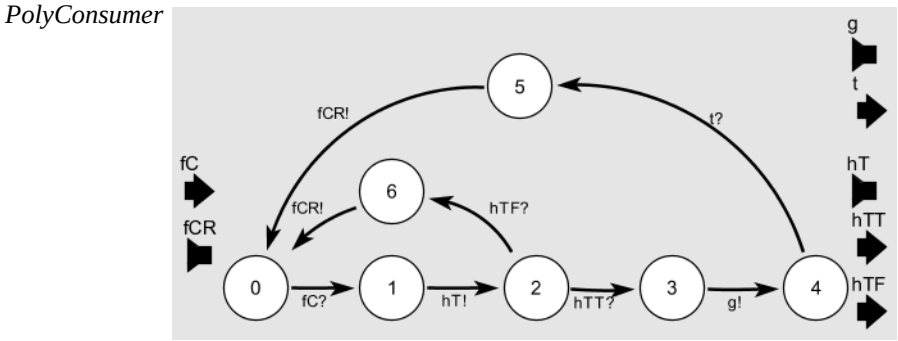


Figure 15. Interface automaton for a domain-polymorphic consumer actor.

In Ptolemy II, actors interact with the director and the receivers of a domain. In figures 14 and 15, 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 4, 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 16 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, they are abstracted out as an internal transition *p_pR* in the figure. Compared with the *SDFdomain* automaton in one of our earlier reports [LeX01], which has *p* and *pR* as separate input and output transitions, the automaton in figure 16 is the projection of the automaton in [LeX01] onto *SDFConsumer*, with the internal transitions combined. 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 17. 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 16 and 17 also reflect the fact that both of the SDF and the DE domains have a

SDFDomain

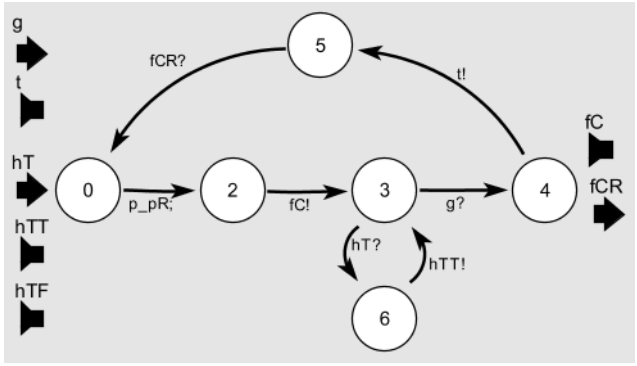


Figure 16. Type signature of the SDF domain.

DEDomain

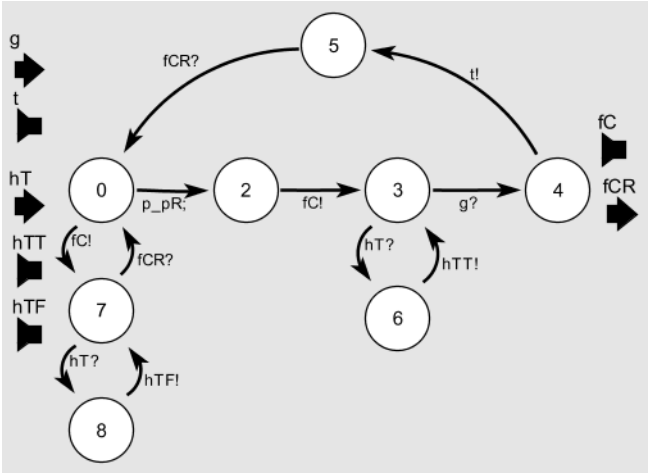


Figure 17. Type signature of the DE domain.

[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.

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 18 and 19 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 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, 10. The case where put() is called before get() is modeled by the transitions among the states 1, 6, 8, 9, 10.

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, 5, 10 in figure 19, while the case where put() is called first is modeled by the transitions among the states 1, 6, 9, 10.

CSPDomain

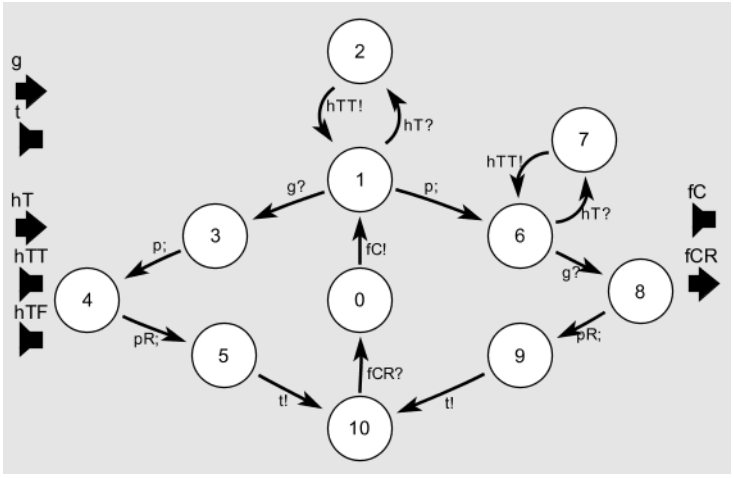


Figure 18. Type signature of the CSP domain.

PNDomain

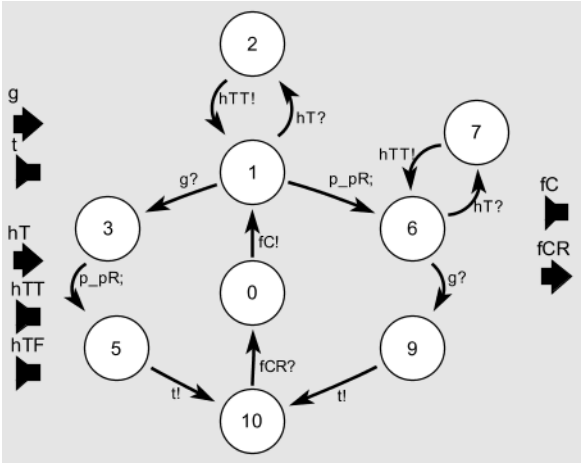


Figure 19. Type signature of the PN domain.

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 5.3.

5.2 Behavioral-Level Type Order and Polymorphism

If we compare the domain automata described in the previous section, 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, from PN to DE, and from CSP to DE. Also, there are alternating simulation relations between any pair of automata among SDF, PN, and CSP, in any direction. This is shown by the directed graph in figure 20. From a type system point of view, the alternating simulation relation denoted in this figure is the subtyping relation. For example, SDF is a subtype of DE, and SDF and PN are subtypes of each other. This subtyping relation can help us design actors that can work in multiple domains. According to the theorem in section 2.3, if an actor is compatible with a certain domain *D*, then the actor is also compatible with the subtypes of *D*. Therefore, this actor is domain polymorphic.

In this formulation, the subtyping relation is not anti-symmetric. That is, two distinct types can be subtypes of each other. This is different from some other type system, such as the data type system in Ptolemy II [XiL00]. When the subtyping relation is anti-symmetric, the subtyping relation induces a partial order. But we do not have a partial order in figure 20. However, if we combine the strongly connected components (SCC) into one node, the component graph becomes a partial order. In figure 20, there is one SCC consisting of SDF, PN, and CSP. The partial order induced by the component graph is shown in figure 21. In this figure, we also added a top and a bottom element. They represent possible domain behavior in extreme cases. One possible design of these two automata is shown in figure 22. In this figure, both automata have a single state. The *BOTTOM* automaton has all the input transitions, and the *TOP* automaton has all the output transitions. We will discuss these two automata further in section 6.3.

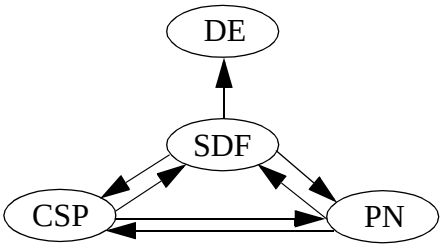


Figure 20. A directed graph showing the alternating simulation relation among domain type

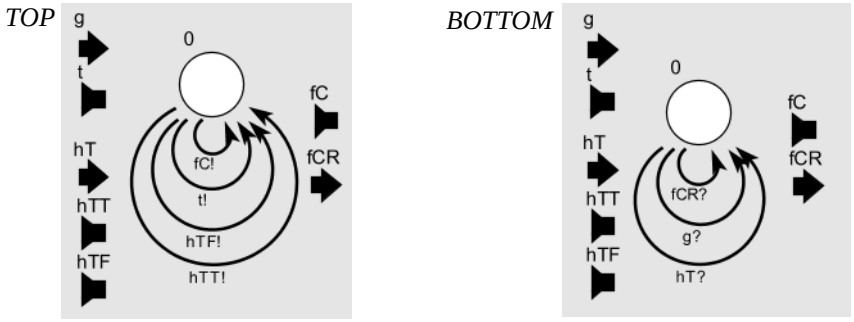


Figure 22. Top and Bottom of the behavioral type order.

When studying the compatibility with actors, the behavioral type order gives a good description for the relation among various behavioral types. From figure 21, it is evident that if an actor is compatible with DE, it is also compatible with any of SDF, PN, and CSP. Also, the *TOP* 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.

5.3 Type Checking Examples

Let's perform a few type checking operations using the actors and domains in the earlier sections. To verify that the *SDFConsumer* in figure 14 can indeed work in the *SDFDomain*, we compose it with the *SDFDomain* automaton in figure 16. The result is shown in figure 23. As expected, the composition is not empty so *SDFConsumer* is compatible with *SDFDomain*.

Now let's compose *DEDomain* with *SDFConsumer*. The result is an empty automaton shown in figure 24. 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 that leads to this condition

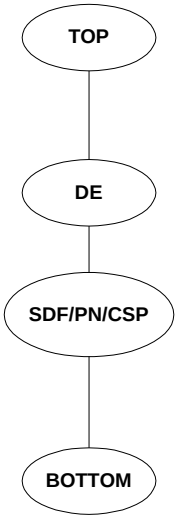


Figure 21. An example of behavioral-level type order.

SDFDomain_SDFConsumer

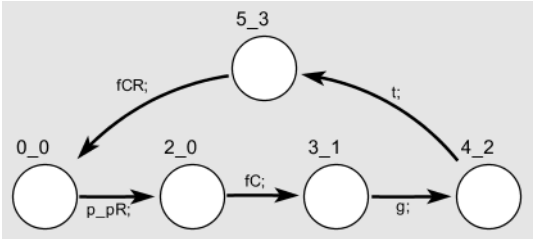


Figure 23. Composition of *SDFDomain* in figure 16 and *SDFConsumer* in figure 14.

DEDomain_SDFConsumer

Figure 24. Composition of *DEDomain* in figure 17 and *SDFConsumer* in figure 14.

is the following: first, both automata take a shared transition *fC*. In this transition, *DEDomain* moves from state 0 to state 7, and *SDFConsumer* moves from state 0 to state 1. At state 1, *SDF-Consumer* issues *g*, but this input is not accepted by *DEDomain* at state 7. So the pair of states (7, 1) in *DEDomain⊗SDFConsumer* 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 consumer cannot be used in the DE Domain.

The *PolyConsumer* in figure 15 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 25. Since *SDFDomain* is below *DEDomain* in the behavioral type order of figure 21, we have also verified that *PolyConsumer* can work in the SDF domain. Therefore, *PolyConsumer* is domain polymorphic. As a sanity check, we have composed *SDFDomain* with *PolyConsumer*, and the result is shown in figure 26.

We have also checked that *PolyConsumer* and *SDFConsumer* are compatible with *CSPDomain* and *PNDomain*. 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 way that many of these actors consume and process tokens can be modeled by the *PolyConsumer* automaton.

5.4 More Detailed Models for Ptolemy II Domains

In the previous sections, the domain automata are designed at a fairly abstract level. That is, they model the combined behavior of a director and a receiver, and they only have the interface to the consumer actor exposed. If we want to model the domains in a little more detail, we can model the directors and receivers separately, and explicitly expose the producer interface. When we do so,

DEDomain_PolyConsumer

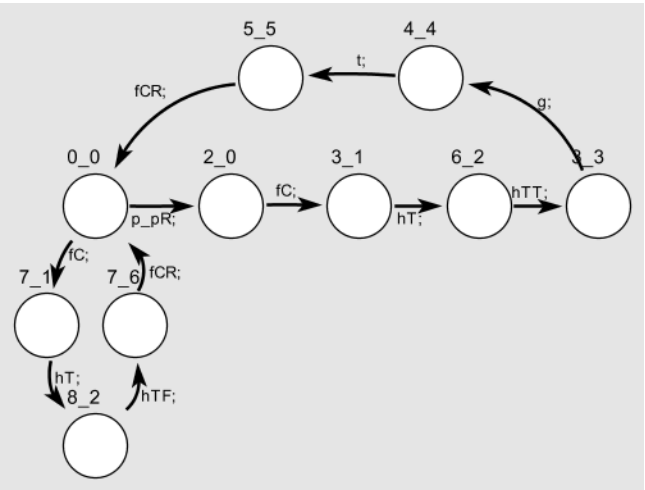


Figure 25. Composition of *DEDomain* in figure 17 and *PolyConsumer* in figure 15.

SDFDomain_PolyConsumer

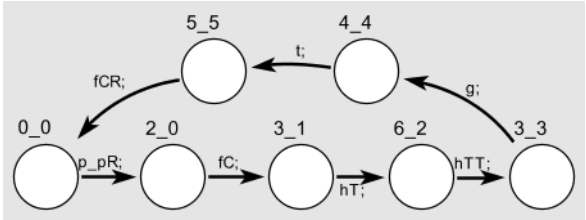


Figure 26. Composition of *SDFDomain* in figure 16 and *PolyConsumer* in figure 15.

we will have more than two automata in the system, so we will need to have transient states in the model.

Starting from the SDF again, figure 27 shows an SDF director for the producer/consumer model in figure 13. Here, *fP* and *fPR* represent the firing of the producer actor and the return of this fire() call. Obviously, the firing schedule in this simple model is just to fire the producer, followed by the consumer, then repeat the cycle indefinitely.

Figure 28 shows an SDF receiver. This receiver is more general than the one described in *SDFDomain* in figure 16 in that it can hold multiple tokens. In particular, at state 2, where a put() call is just returned, the receiver allows another put() call to come before get() is called. This is modeled by the transition *p* from state 2 to state 1. Also, after a get() call, the receiver may not be empty, so a transition *t* from state 3 may take the receiver back to state 2. Notice that this automaton is non-deterministic, and it uses non-determinism to allow more flexible ordering between the put() and get() calls without explicitly modeling the number of tokens in it. Also notice that states 1t, 3t, 4t, and 5t are transient, for reasons similar to the transient states in figure 7.

The composition of *SDFDirector* and *SDFReceiver* represents the behavior of the SDF domain. This composition can be composed with a producer actor and a consumer actor. Figure 29 describes the behavior of a typical producer actor. It simply calls put() in its fire() method. As

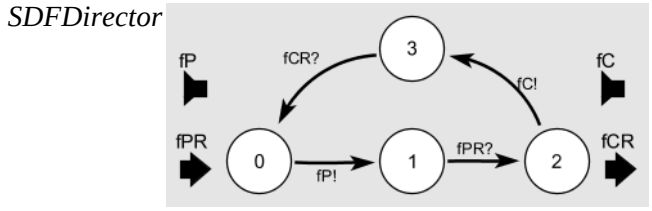


Figure 27. An SDF director.

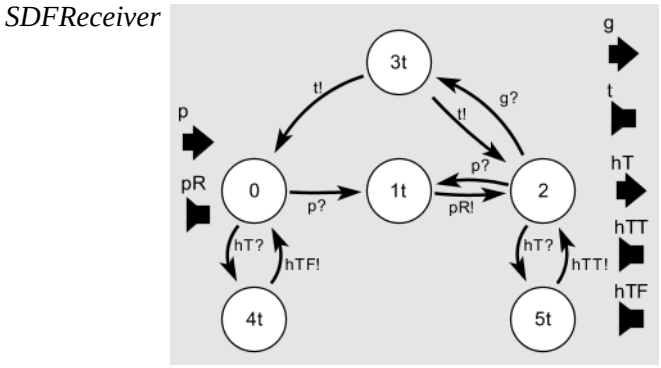


Figure 28. An SDF receiver.

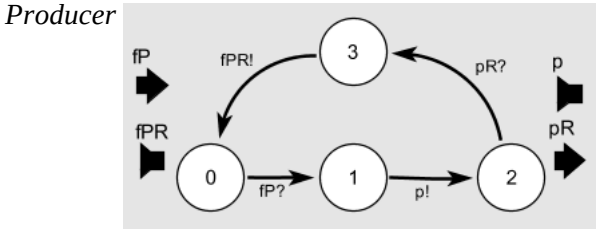


Figure 29. A producer.

a type checking example, we can compose *SDFDirector*, *SDFReceiver*, *Producer*, and *SDFConsumer* together. We omit this composition here for the sake of brevity.

Now let's look at the DE domain. Figure 30 and 31 show a DE director and a DE receiver, respectively. Different from the SDFDirector, the DEDirector does not statically schedule the firing of the producer and consumer. Also, since actor execution in DE is scheduled based on the time events occur, a token put into a receiver may not be immediately available for the consumer until the simulation time reaches the time of the token, so we have a transition *pR* from state 1 to 0 in *DEReceiver*.

If we want to check the subtyping relation between SDF and DE, we can use the compositional property of alternating simulation [deH01] to simplify the checking. According to this property, if *SDFDirector* is a subtype of *DEDirector*, and *SDFReceiver* is a subtype of *DEReceiver*, we have the composition of *SDFDirector* and *SDFReceiver* to be a subtype of the composition of *DEDirector* and *DEReceiver*. Indeed, we have verified the relation between the directors and receivers, so we know that above result holds. This is shown in figure 32.

DEDirector

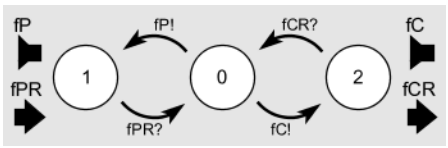


Figure 30. A DE director.

DEReceiver

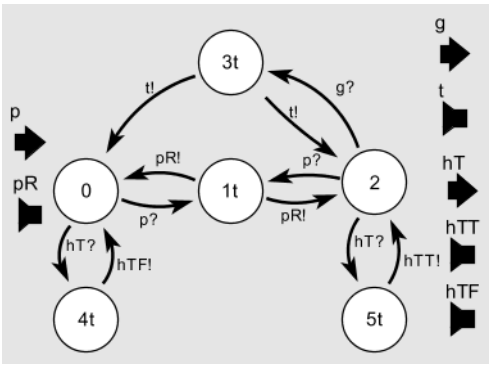


Figure 31. A DE receiver.

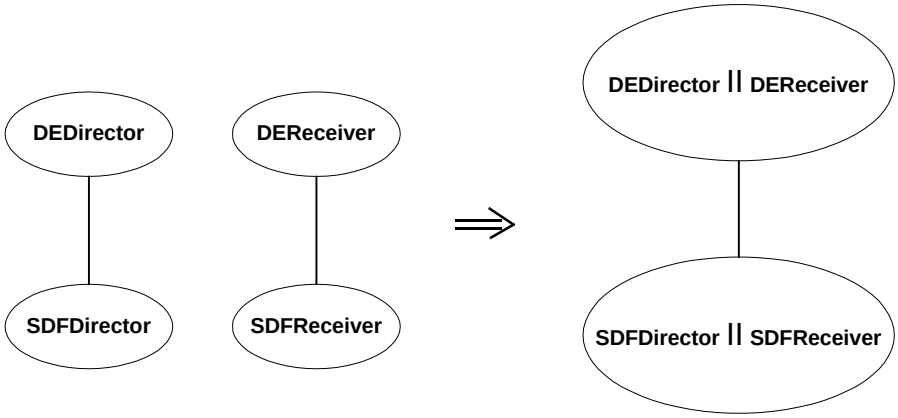


Figure 32. Using the compositional property to show that *SDFDomain* is a subtype of *DEDomain*.

For the producer/consumer model, the director in the PN domain will create two threads to run the producer and consumer, as shown in figure 33(a) and (b). The whole PN director is the composition of these two, as shown in figure 33(c). The receiver for the PN domain is shown in figure 34. It performs blocking read and non-blocking write. That is, if the get() call arrives before put(), it blocks. But the put() call never blocks. This receiver also allows multiple put() calls before get().

If we want to check the subtyping relation between PN and SDF, we can check whether there is an alternating simulation between the directors and receivers of these two domains. Unfortunately, they do not have the same relation as we had with the simple domain models in section 5.2. Here, the only alternating simulation we have is one from the *SDFDirector* to *PNDirector*. This example shows that the subtyping relation depends on the design of the domain automata. In the *SDFReceiver* automaton in figure 28, the hasToken() call returns *false* at state 4t, while the *PNReceiver* in figure 34 returns *true* in the same state. This difference breaks the alternating simulation relation. However, this lack of alternating simulation does not mean that an SDF actor cannot work in the PN domain. In fact, with respect to the communication protocol, any SDF actor can work in the PN domain. It is just that the alternating simulation does not capture this relation.

Although there is no alternating simulation from *PNDirector* to *SDFDirector*, there is actually an alternating simulation when these automata are projected to the producer or consumer automata. Figure 35(a) and (b) show the projection of *SDFDirector* and *PNDirector* onto the *Producer* in figure 29. We can verify that (1) There is an alternating simulation from *PNDirectorToProducer* to *SDFDirectorToProducer*; (2) *SDFDirectorToProducer* is compatible with Producer. So

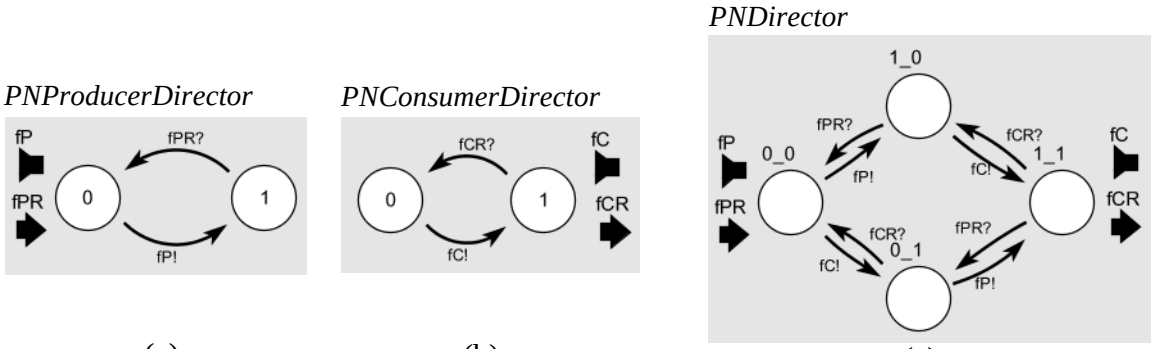


Figure 33. PN director.

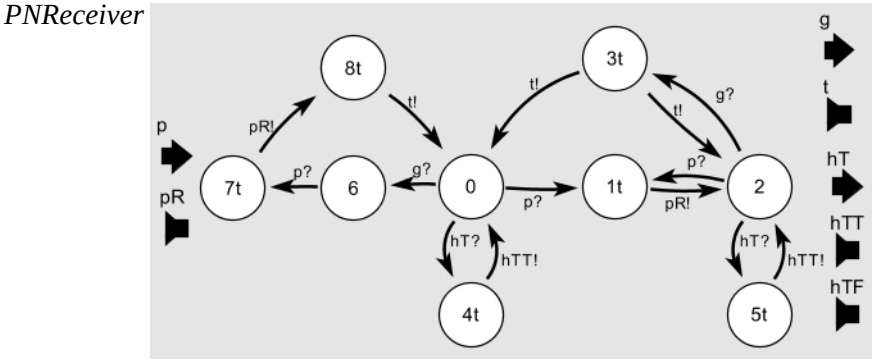


Figure 34. A PN receiver.

according to the result in section 3.2, we know that *PNDirector* is compatible with *Producer*. This example shows that the projection automata can be used to expose some alternating simulation relations when the original automata do not have this relation.

We skip the director and receiver automata for the CSP domain. The CSP director automaton is the same as the *PNDirector* in figure 33, and the CSP receiver performs both blocking read and blocking write.

6. Discussion

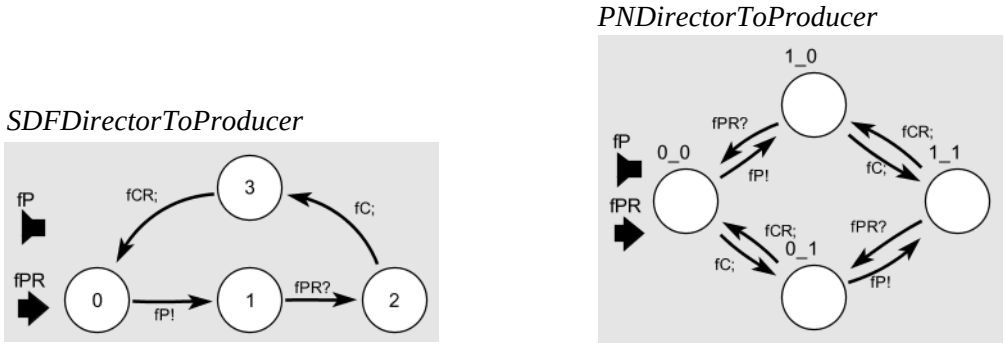
6.1 Reflection

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 *SDFConsumer* of figure 14 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.*

6.2 Trade-offs in Type System Design

The examples in sections 5.1 and 5.4 show that there is no canonical type representation because behavioral types can be specified at different abstraction levels. These examples focus on the communication protocol between one or two actors and their environment. This scope can be broadened by including the automata of more actors and using an even more detailed domain 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.

The previous examples also show that the subtyping relation among domain types can help simplify type checking. However, because the alternating simulation relation is sensitive to the design of the domain automata, we do not always have the same subtyping relations in different design. In some cases, such as in section 5.4, we can increase the set of relations captured by using



(a) (b)

Figure 35. Projection of *SDFDirector* and *PNDirector* to *Producer*.

the projection automata, but this is not always possible. So the trade-off between the amount of information carried by the types and the amount of subtyping relations in the system also need to be considered.

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 behavioral types. In this space, one extreme point is complete 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 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 (states without any outgoing transitions) 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 behavioral 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 5, is a good starting point.

6.3 Top and Bottom

We have shown one possible design for the top and bottom elements of the behavioral type order in figure 22. These two automata are very general in that they are not only the top and bottom elements of the partial order in figure 21, but also the top and bottom of the partial orders formed by any set of automata with the same set of input and output transitions. In another word, there is an alternating simulation relation from any automaton to the *TOP* automaton in figure 22, and an alternating simulation relation from the *BOTTOM* automaton in figure 22 to any automaton with the same inputs and outputs.

If we can design an actor that is compatible with the *TOP* automaton, then that actor will be maximally polymorphic in that it will be able to work in any domain that may be created. However, it is easy to see that this is almost impossible. Since the *TOP* automaton may issue any output at any time, no non-trivial actor can be compatible with it. This means that we cannot hope to design a non-trivial actor that will be able to work in any environment.

On the other hand, the *BOTTOM* automaton is compatible with any actor automaton. For example, the compositions of *BOTTOM* with the *SDFConsumer* or the *PolyConsumer* are shown in figure 36. The two compositions are the same. Intuitively, since the *BOTTOM* automaton does not have any output transition, it does not call the `fire()` method of the actor, so there is no interaction between the *BOTTOM* automaton and the actor automaton.

The *TOP* and *BOTTOM* automata represent two extremes of the possible environments for actors. The *TOP* is the most stringent environment in which no non-trivial actor can work, while *BOTTOM* is the laxest environment in which an actor is not asked to do anything.

6.4 Related Work

6.4.1 Behavioral Types

In the concurrent object-oriented language community, there is a lot of ongoing work on type systems for parallel object languages and calculi. Some of the proposed systems have very similar objectives as ours, namely, capturing the dynamic behavior of components. In particular, the type model of Puntigam [Pun96] and the behavioral type system of Najm and Nimour [NaN99][NNS99] both attempt to capture the communication behavior of components, and both systems have a notion of subtyping that is conceptually similar to the alternating simulation relation.

The type model of Puntigam is designed for a language that is based on a combination of the actor model [Agh86] and a process calculus with trace semantics. Similar to our model, objects communicate by message passing. A message has the form $c(o_1, ..., o_m; v_1, ..., v_n)$. This can be viewed as a method call with method name c, input parameters $o_1, ..., o_m$, and output parameters $v_1, ..., v_n$. A *type trace* is a sequence $p_1 ... p_n$ of message prototypes, and a *type trace set T* is a prefix-closed non-empty set of type traces. Here, the type trace sets are the type specifications of active objects. It defines the sequences of messages that an object is prepared to handle, and the clients of the objects are allowed to send message only according to exactly one type trace selected from the set. The type trace set of a type τ is denoted trace(τ). Under this formulation, a subtype is defined as:

A type σ is a subtype of a type τ (denoted by $\sigma \leq \tau$) if and only if for each type trace $p_1 ... p_n \in$ trace(τ) there is a $p_1' ... p_n' \in$ trace(σ) so that (for each $1 \leq i \leq n$; with $p_i = c_i(\phi_{i,1}, \ldots, \phi_{i,k_i}; \varphi_{i,1}, \ldots, \varphi_{i,l_i})$ and $p'_i = c'_i(\phi'_{i,1}, \ldots, \phi'_{i,k'_i}; \varphi'_{i,1}, \ldots, \varphi'_{i,l'_i})$

- $c_i = c'_i$ (equal message identifiers);

BOTTOM_SDFConsumer or *BOTTOM_PolyConsumer*

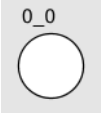


Figure 36. Composition of BOTTOM and *SDFConsumer* or *PolyConsumer*.

- $k'_i \le k_i$ and $\phi_{i,j} \le \phi'_{i,j}$ for $1 \le j \le k'_i$ (contravariant input parameter types);
- $l_i \le l'_i$ and $\varphi'_{i,j} \le \varphi_{i,j}$ for $1 \le j \le l_i$ (covariant output parameter types).

Similar to the alternating simulation relation we use for defining subtypes in our system, this definition has contravariant input types and covariant output types.

However, there are several differences between this formulation and ours. First, a trace is a global property in that a trace specifies a complete run of an object, while the simulation relation is local in that it is a relation for each step of the run. Second, the subtyping definition here mixes data typing issues with behavior. In particular, the last two conditions in the definition is essentially the standard record subtyping rules in many data type systems [CAR97]. In our system, we separate the data typing issues from behavioral typing and handle them in different ways. Third, the trace set, which defines a language, is more general than an automaton. If the trace set is constrained to be a regular set, then it is equivalent to an automaton.

The behavioral type system presented in [NaN99] is closer to our system. This system is designed for an object calculus which is a variant of the π-calculus [MPW92], with syntactic sugar for method definition. Here, behavioral types specify the set of methods (services) an objects supports. This set is dynamic since the set of supported methods may change after each method call. For example, an object implementing a one place buffer has a `put()` and a `get()` method for writing and reading data into and from the buffer. When the buffer is empty, the set of supported methods includes only the `put()` method. After `put()` is called, the set includes only the `get()` method, and so on. This dynamic behavior is specified using a labeled transition system, where each transition is a method signature. Similar to our system, the definition of subtyping distinguishes the sending and receiving of messages. If a type X_2 is a subtype of a type X_1, then all the receiving actions of X_1 can be performed by X_2, and all the sending actions of X_2 can be performed by X_1. This is analogous to the alternating simulation relation of interface automata. The formal definition of behavioral types, the transition system, and subtyping can be found in [NaN99][NNS99]. In this system, the requirements for type compatibility are defined by complicated type rules.

In both of the above two systems, the basic goal of typing is to ensure that an object does not receive a method call that is not supported. This error condition is analogous to the error condition that results in illegal states in the composition of interface automata. Compared with them, our interface automata based system permits much easier type checking. Also, since interface automata can be easily described in bubble and arc diagrams, the type representation in our system is easier to understand than the algebraic form used in both approaches. Another difference is that the above two systems concentrate mostly on the communication between objects through message passing, while our system also takes the execution control into consideration. Finally, it is interesting to note the different terminologies used to describe the dynamic behavior of components. Inspired by [NaN99], we call our description behavioral types, while it is called process types in [Pun96], and we ourselves had previously called it system-level types [LeX01].

6.4.2 Component Interfacing

In hardware design, many people have proposed techniques of *protocol synthesis* to connect components with different interfaces [COB92][COB95][EiP00][ETT98][OrB97][PRS98]. There are two approaches to protocol synthesis. One is library or template based. For example, Eisenring and Platzner [EiP00] develop a tool that provides a template and a corresponding generator

method for each interface type. The other is to generate a converter from the two interfaces to be connected. For example, Passerone *et al.* [PRS98] describe the communication protocols of the two components to be interfaced by two finite state machines, and the converter is essentially the product machine, with invalid states removed. Compared with this approach of component interfacing, our approach is to design polymorphic components with tolerant interfaces, so that they can be used in different settings. Besides, there are two additional differences between our system and the protocol synthesis techniques.

One difference is that behavioral types cover multiple models of computation, while protocol synthesis usually concentrates on interfacing different implementations of one model of computation. For example, Passerone *et al.* [PRS98] focus on synchronous model (shared clock); Eisenring and Platzner [EiP00] study dataflow models implemented by queues between component; in [ETT98], Eisenring *et al.* design a system using synchronous dataflow; and in [OrB97], Ortega and Borriello use a communication protocol with a non-blocking write behavior, which is similar to the one in process networks.

Another difference is on the level of abstraction. Since design is a process of refinement, the description of a component may exist at different levels. In [ETT98], Eisenring, *et al.* divide the possible abstractions into two levels: *abstract communication types* and *physical communication types*. Abstract communication types includes buffered versus non-buffered, blocking versus non-blocking, synchronous versus asynchronous communication. Physical communication types includes memory-mapped I/O, interrupt or DMA-transfer. In [BLO98], Borriello, *et al.* gave a more contiguous categorization of interface levels: electrical, logical, sequencing, timing, data transaction, packet, and message. The behavioral type work addresses the highest level in this classification: different mechanisms for message passing. It covers the abstract communication types. On the other hand, most work on protocol synthesis is at the hardware or architecture levels. For example, reconfigurable computing with FPGA is targeted in [EiP00]; [PRS98] is about RTL level interface synthesis; the problem of mapping a high-level specification to an architecture is considered in [OrB97]; and a system to generate interface between a set of microprocessors and a set of devices is described in [COB92][COB95].

The differences between our type system and the work in protocol synthesis make them complementary to each other. They may be used at the different stages of the design process.

7. Conclusion and Future Work

We have proposed two extensions to the interface automata formalism. Transient states allow us to model mutual exclusion easily, and projection automata expose alternating simulation between two automata for a subset of their interfaces.

Based on the extended interface automata, we have described a type system that captures the interaction dynamics 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 behavioral 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 this 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 behavioral 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. The discussion with Winthrop Williams was very helpful. 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

[Agh86] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1986.

[BCD02] S. S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java," *Technical Memorandum UCB/ERL M02/23*, EECS, University of California, Berkeley, August 5, 2002. (http://ptolemy.eecs.berkeley.edu/publications/papers/02/ptIIdesign/)

[BLO98] G. Borriello, L. Lavagno, and R. B. Ortega, "Interface Synthesis: A Vertical Slice from Digital Logic to Software Components," Proc. of International Conference on Computer Aided Design (ICCAD), San Jose, CA, USA, Nov. 8-12, 1998.

[CAR97] L. Cardelli, "Type Systems," *The Computer Science and Engineering Handbook*, CRC Press, 1997.

[CaW85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, Vol.17, No.4, Dec. 1985.

[COB92] P. Chou, R. B. Ortega and G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems," *Proc. ICCAD*, pp488-495, Nov. 1992.

[COB95] P. Chou, R. B. Ortega and G. Borriello, "Interface Co-Synthesis Techniques for Embedded Systems," *Proc. of the Int. Conf. on Computer Aided Design*, Nov. 1995.

[DaP90] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[deH01] 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.

[EiP00] M. Eisenring and M. Platzner, "Synthesis of Interfaces and Communication in Reconfigurable Embedded Systems," *IEE Proc. Comput. Digit. Tech*, 147(3), May 2000.

[ETT98] M. Eisenring, J. Teich and L. Thiele, "Rapid Prototyping of Dataflow Programs on Hardware/Software Architectures," *Proc. 31st Annual Hawaii International Conference on System Sciences*, 1998.

[Hoa78] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, 28(8), August 1978.

[Kah74] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[KaM77] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[Lee02] E. A. Lee, "Embedded Software," to appear in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[LeM87] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, Sept., 1987.

[LeV01] E. A. Lee and P. Varaiya, Structure and Interpretation of Signals and Systems, Book Draft, U.C.Berkeley, 2001. (http://ptolemy.eecs.berkeley.edu/eecs20/supplements/master.pdf)

[LeX00] E. A. Lee and Y. Xiong, "System-Level Types for Component-Based Design," *Technical Memorandum UCB/ERL M00/8*, EECS, University of California, Berkeley, Feb. 29, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/systemLevel/)

[LeX01] E. A. Lee and Y. Xiong, "System-Level Types for Component-Based Design," *First Workshop on Embedded Software*, EMSOFT2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001.

[LyT81] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th ACM Symp. Principles of Distributed Computing*, pp 137-151, 1981.

[Mit84] J. C. Mitchell, "Coercion and Type Inference," *11th Annual ACM Symposium on Principles of Programming Languages*, 175-185, 1984.

[NaN99] E. Najm, A. Nimour, "Explicit Behavioral Typing for Object Interface," *Semantics of Objects as Processes, ECOOP'99 Workshop*, Lisbon, Portugal, June, 1999.

[NNS99] E. Najm, A. Nimour and J.-B. Stefani, "Infinite Types for Distributed Object Interfaces," *Third IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, Firenze, Italy, Feb., 1999.

[MPW92] R. Milner, J. Parrow, D. Walker, "A Calculus of Mobile Processes (Part I and part II)," *Information and Computation*, 100:1-77, 1992.

[Ode96] M. Odersky, "Challenges in Type System Research," *ACM Computing Surveys*, 28(4), 1996.

[OrB97] R. B. Ortega and G. Borriello, "Communication Synthesis for Embedded Systems with Global Considerations," *Proc. of the 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE'97)*, March 1997.

[PRS98] R. Passerone, J. A. Rowson and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," *35th Design Automation Conference*, 1998.

[Pun96] F. Puntigam, "Types for Active Objects Based on Trace Semantics," *Proc. of the Workshop on Formal Methods for Open Object-Oriented Distributed Systems (FMOODS'96)*, Paris, France, March, 1996.

[XiP98] 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.

[Xio02] Y. Xiong, "An Extensible Type System for Component-Based Design," Ph.D. Thesis, *Technical Memorandum, UCB/ERL M02/13*, University of California, Berkeley, CA 94720, May 1, 2002.

[XiL00] 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.