DOI 10.1007/s00165-004-0043-8
BCS © 2004
Formal Aspects of Computing (2004) 16: 210–237

Formal Aspects
of Computing

A behavioral type system and its application in Ptolemy II

Edward A. Lee[1] and Yuhong Xiong[2]

[1] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, USA
[2] Hewlett-Packard Laboratories, Palo Alto, CA, USA. Email: yuhong.xiong@hp.com

Abstract. Interface automata [deH01] have been introduced as an interface theory [deH01a] capable of functioning as a behavioral type system. Behavioral type systems describe dynamic properties of components and their compositions. Like traditional (data) type systems, behavioral type systems can be used to check compatibility of components. In this paper, we use interface automata to devise a behavioral type system for Ptolemy II, leveraging the contravariant and optimistic properties of interface automata to achieve behavioral subtyping and polymorphism. Ptolemy II is a software framework supporting concurrent component composition according to diverse models of computation. In this paper, we focus on representing the communication protocols used in component communication within the behavioral type system. In building this type system, we identify two key limitations in interface automata formalisms; we overcome these limitations with two extensions, transient states and projection automata. 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, which enable dynamic component creation, morphing application structure, and admission control. We discuss the trade-offs in the design of behavioral type systems.

Keywords: Behavioral types; Behavioral subtyping; 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, concurrent component-based design is established as an important approach to handle complexity. Many tools have been developed to support this design methodology, including for example Simulink (from The MathWorks), SPW (the Signal Processing Worksystem, from Cadence), CoCentric System Studio (from Synopsys), Metropolis [GoS02], and Ptolemy II [BCD02]. In such tools, components are (at least conceptually) concurrent, and interact by sending messages according to some communication protocol. Such component-based design has been termed *actor oriented* [Lee02], to distinguish it from object oriented, where components (in practice) interact by method calls. The communication protocols and concurrency policies together are called the *model of computation*.

Correspondence and offprint requests to: Edward A. Lee, EECS, University of California, Berkeley CA 94720, USA. Email: eal@eecs.berkeley.edu

In any form of component-based design, 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 concentrates on data types, and leaves the checking of dynamic behavior to other techniques.

Largely outside the component-based design community, type systems that capture various dynamic properties of programs have been studied under various settings, such as concurrent ML [NiN94], actor-based languages [CPS97], object-oriented subtyping [LiW94], π-calculus and similar process algebras [IgK01] [KPT96] [NaN99] [NNS99] [PiS93] [Pun96] [RaR02], the CHAM formalism [IWY00], architectural description languages [AlG97], and popular languages like C [STS01] and Java [LBR98][LNS00]. However, these systems do not address the challenges of component-based design directly, either because the properties they capture are not critical for component-based design, or because they are too abstract to be implemented in the existing tools directly.

In this paper, we present a practical behavioral type system that captures the dynamic aspects of component interaction. 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. Interface automata have two key strengths for our purposes. First, their composition semantics uses an "optimistic" approach that leads to simple compositions and straightforward compatibility checking. Second, unlike most automata formalisms, they treat inputs and outputs differently, in a manner analogous to the co/contra-variance relation in function subtyping; this enables a useful form of subtyping that permits the definition of behaviorally polymorphic components. A third, less critical strength is that the visual representation of interface automata is more accessible than the algebraic notations used in many methods, although the latter are often more compact.

Traditionally, automata models are used to perform model checking at design time. Here, our emphasis is not on model checking to verify arbitrary user-defined behavior, but rather on compatibility of the composition of pre-defined types. In object-oriented type systems, scalability is ensured by building composite types from a pre-defined set of primitive types. Users do not extend the set of primitive types (typically). We similarly propose to define a set of primitive behavioral types that captures key properties of models of computation and the components that operate within them. Much as an object-oriented type system does not capture all aspects of the static structure of an application, our behavioral type system will not capture all aspects of the dynamic behavior. If we design it well, however, it will capture enough aspects to greatly improve design. 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 on the Ptolemy II component-based design environment [BCD02]. This platform includes a visual editor for defining behavioral types and a suite of tools for composing and analyzing these type definitions. All graphics in this paper are taken from this visual editor. All compositions have been computed using its tools, and all subtyping relations have been checked using its tools.

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 extent. 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 $Int \leqslant Double$, 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 is 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. Interface automata have an alternating simulation relation. This relation is analogous to the co/contra-variance relation in function subtyping, and is natural for the subtyping relation in our system. Using this relation, 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, and sometimes even impossible, 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 components are connected. 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 include these extensions and some experimental results 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 were devised for capturing the temporal aspects of software component interfaces [deH01]. As with other automata models, interface automata consist of states and transitions, 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

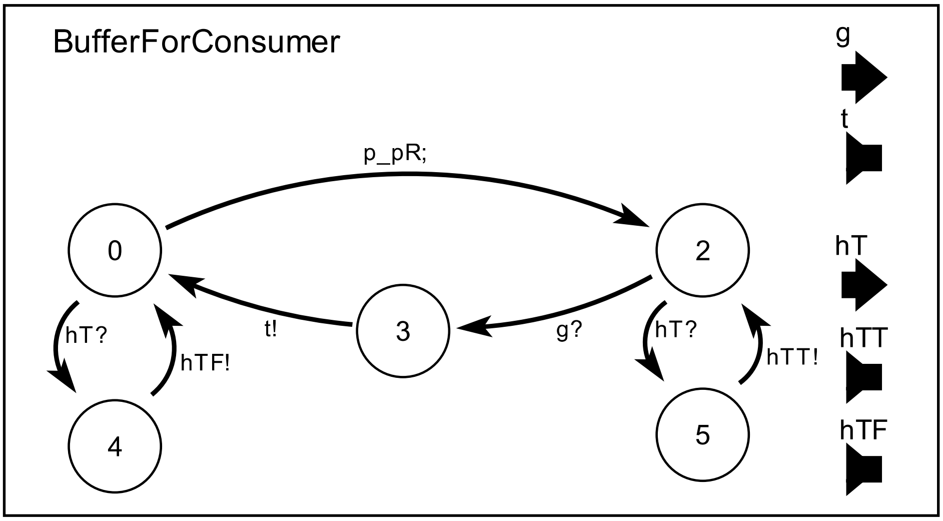

Fig. 1. Interface automaton model for a one-place buffer (only the consumer interface is modeled)

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 the producer in later examples. Figure 1 shows the interface automaton model for the buffer. This and the subsequent figures are drawn in the Ptolemy II software [BCD02], within which we have built a "domain" for representing, composing, analyzing, and manipulating interface automata.

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 Fig. 1 denote the inputs and outputs of the automaton. They are:

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

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. This characteristic makes them light-weight. In Fig. 1, the input g is only accepted at state 2, but not in any other states. This is different from I/O automata [LyT81], which is syntactically similar to interface automata. 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 Fig. 1, we do not have states and transitions to describe the case when get() is called on an empty buffer.

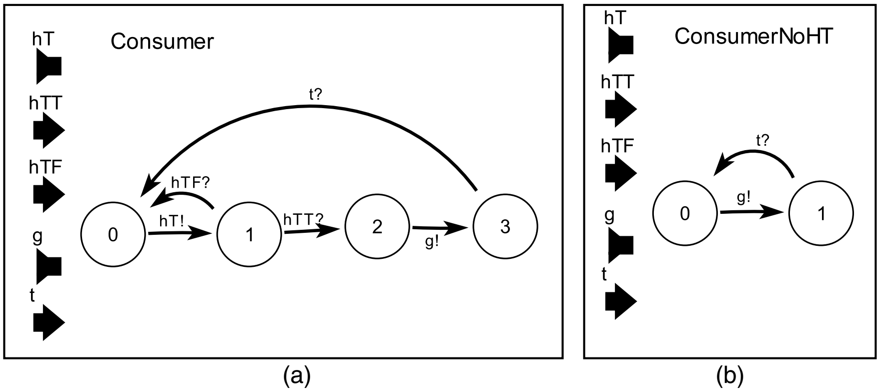

Fig. 2. Two consumer automata

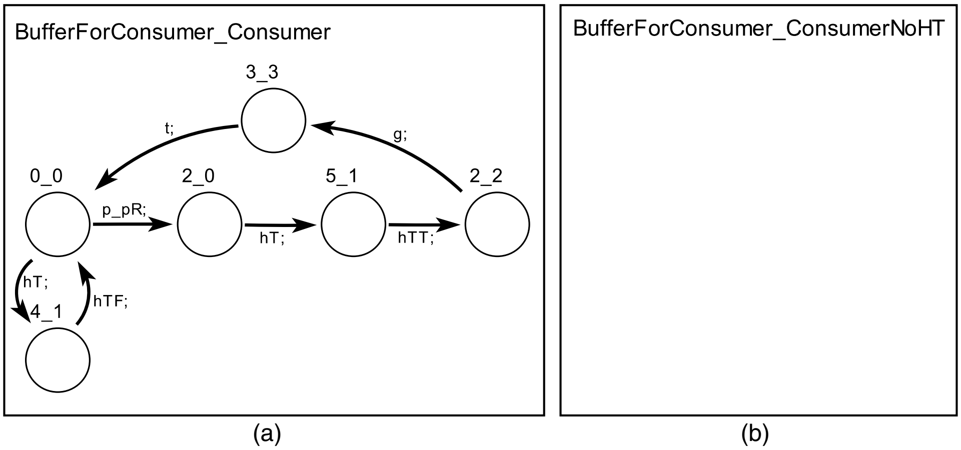

Fig. 3. Composition of *BufferForConsumer* in Fig. 1 and the two consumer automata in Fig. 2

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. Internal states are like the τ-step in CCS, or concealed states in CSP.

Figure 2 shows two consumer automata that can be composed with the automaton *BufferForConsumer* in Fig. 1. The *Consumer* automaton in Fig. 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 Fig. 3(a). In Fig. 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 Fig. 3(b). This is a key property of interface automata. More conventional automaton composition always results in a state space that

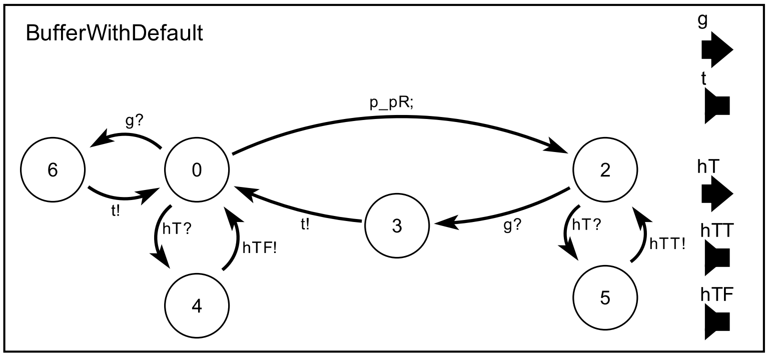

Fig. 4. A buffer that can return a default token

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 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. Although the guarantee provided by the optimistic approach appears weaker, this approach is more natural for design purposes, because components are usually designed under assumptions about the environment. Also, the authors of interface automata pointed out that the composition of interface automata is often smaller and thus easier on the user, than the traditional pessimistic models, which must take all kinds of environment into account.

2.3. Alternating simulation

Interface automata have a notion of *alternating simulation*, which, unlike the conventional simulation relation in automata theory, treats inputs and outputs differently to provide a co/contra-variance relation like that found in function subtyping. We use alternating simulation to simplify type checking and to realize behaviorally polymorphic interfaces. Informally, for two interface automata P and Q, there is an alternating simulation relation from Q to P if all the input transitions of P can be simulated by Q, and all the output transitions of Q can be simulated by P. The formal definition, which involves the notions of ϵ-closure and externally enabled inputs and outputs, is given in [deH01]. These notions are for taking into account the fact that the internal transitions of P and Q are independent. Since we do not need this level of detail here, we omit the precise definition. Instead, we just give an example of alternating simulation:

The *BufferWithDefault* automaton in Fig. 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 Fig. 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. If we compare *BufferWithDefault* with *BufferForConsumer*, since all the transitions at the states 2, 3, 4, 5 are the same in them, the requirement for alternating simulation is trivially satisfied. At state 0, all the input transitions of *BufferForConsumer*, which is just $hT?$, can be simulated by *BufferWithDefault*, and all the output transitions of *BufferWithDefault*, which is empty, can be trivially simulated by *BufferForConsumer*. So 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

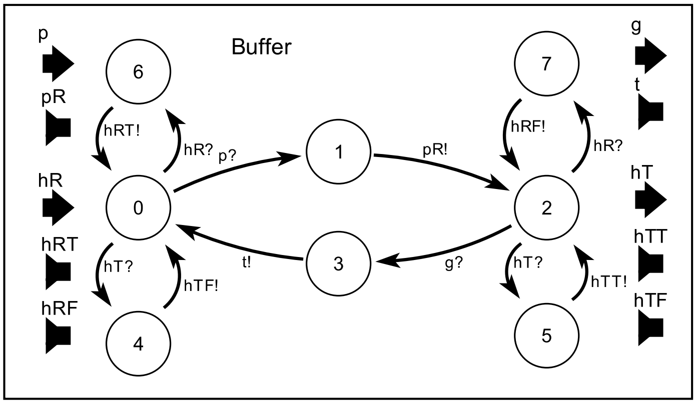

Fig. 5. A one-place buffer

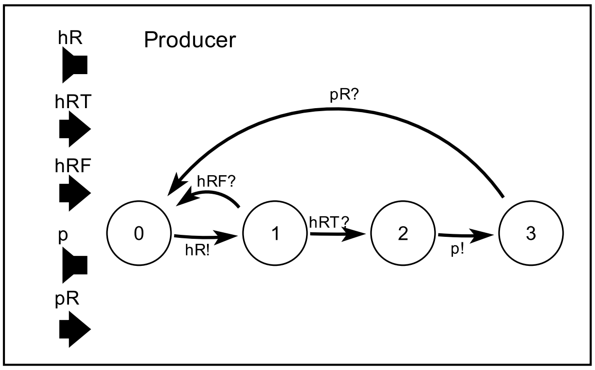

Fig. 6. A producer

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

In constructing our behavioral type system and attempting to use it to represent key properties of Ptolemy II domains and components, we encountered two key obstacles. First, modeling atomic actions that involved more than one transition in an automaton proved extremely awkward, and yet was very commonly needed. Second, the usual way of composing interface automata, which matches inputs and outputs by name, does not provide sufficient scoping constraints to handle practical designs; in particular, composing more than two automata that interact through different interfaces is awkward. We describe here two extensions to interface automata that address these obstacles.

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

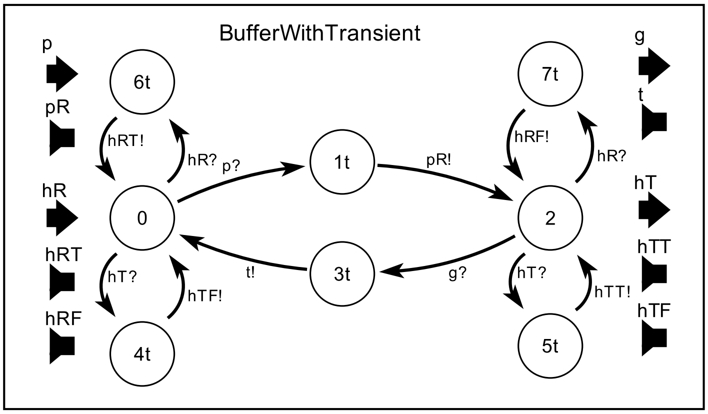

Fig. 7. A buffer with transient states

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 Fig. 5, we assume that its methods execute atomically. 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 are not intrinsically atomic.

It is possible to modify these models to achieve mutual exclusion. For example, if we want to model the synchronization mechanism in Java, we can add an output "lock" and another "unlock," and then 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 very cumbersome. Since such mutexes are so common, we want to support them in the automata formalism. Moreover, they are more commonly needed 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 Fig. 7. Transient states can only have output and internal transitions emanating from them. 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) or takes a transition in response to an input. To illustrate this with a simple example, consider two automata in Figs. 8(a) and 8(b). Without considering transient states, their composition is empty, as shown in Fig. 8(c). This is because that the automaton in (b) may send out b!, which is not enabled at state 0 in the automaton in (a). However, if we treat state 0 in the first automaton as transient, then the composition of the two automata is not empty, as shown in Fig. 8(d). As a bigger example, the composition of the *Producer* in Fig. 6, the *BufferWithTransient* in Fig. 7, and the *Consumer* in Fig. 2(a) is shown in Fig. 9.

We currently do not allow the composition to enter a pair of states where both of them are transient. Relaxing this constraint requires further research.

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, but then we lose the co/contra-variance relation like that found in function subtyping. For example, the FSM model for the buffer is shown in Fig. 10. By adding the notion of transient states to interface automata, we achieve the ability in FSM to model atomic input and output.

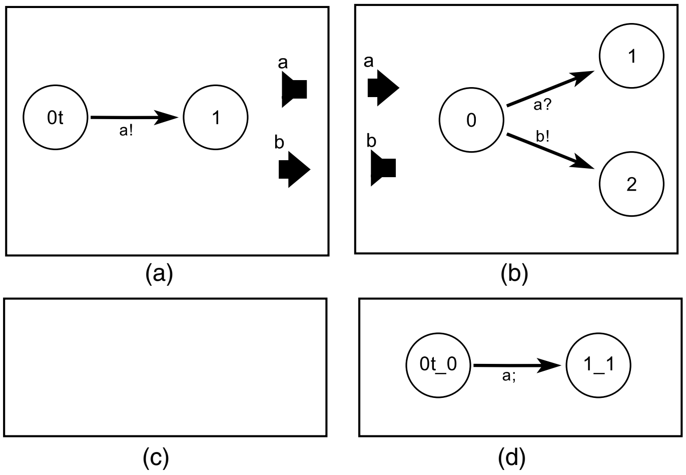

Fig. 8. Example of transient state

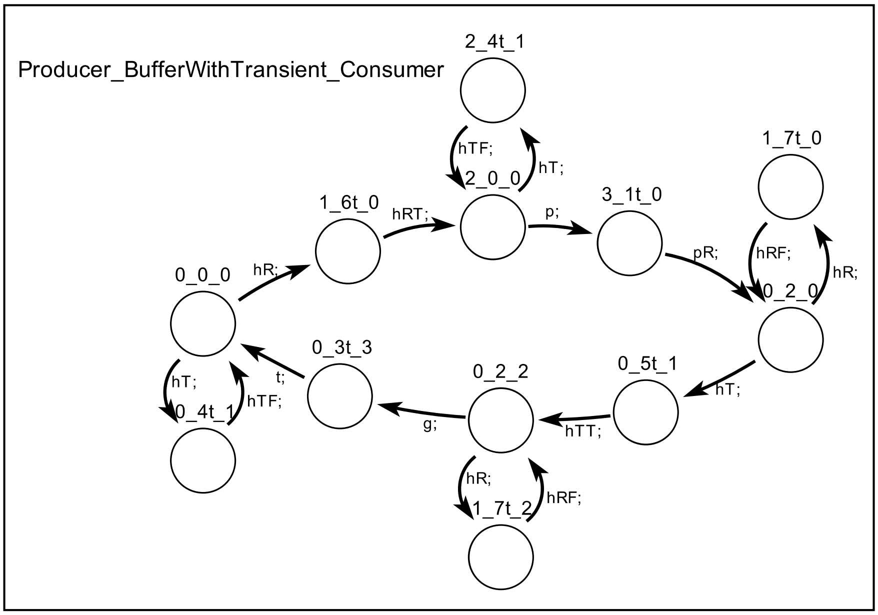

Fig. 9. Composition of the *Producer* in Fig. 6, *BufferWithTransient* in Fig. 7, and *Consumer* in Fig. 2(a)

3.2. Explicit connections and projection automata

In most formalisms for composing automata, including interface automata, an output of one automaton is matched to an input of another by name matching. Unfortunately, there are only weak notions of scoping for such names, so the names, in effect, become global. It is difficult to express that a particular output might be intended for consumption by a particular other automaton, and be intended to be ignored by another automaton in a composition. Thus, we cannot easily distinguish the erroneous case, where one automaton issues an output that the other cannot accept, from the harmless case, where one automaton issues an output that is ignored by the other automaton and becomes an output of the composition. We give an example to illustrate the problems this creates.

In *BufferWithTransient* in Fig. 7, 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.

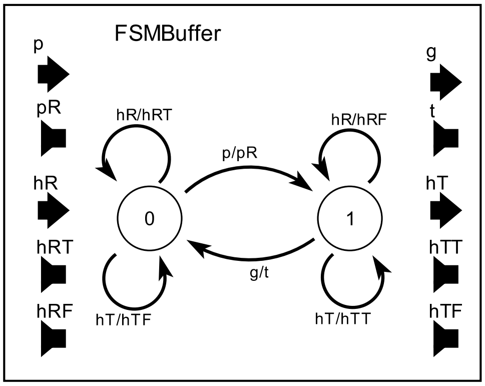

Fig. 10. FSM model for the buffer in Fig. 7

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 Fig. 11. Here, the output *c* represents the count event. It goes to a certain counter component.

Intuitively, *BufferWithCounter* in Fig. 11 should function as a subtype of *BufferWithTransient*. That is, it should compose with any consumer that *BufferWithTransient* can compose with. In interface automata, the subtype relation is the alternating simulation relation. Regrettably, these two automata do not have an alternating simulation between them. If we analyze them carefully, we realize that the additional output *c* in *BufferWithCounter* is "interfering" the alternating simulation. Even though this output does not affect the consumer, it obscures the relation between the two buffer automata with respect to the consumer.

The problem here is weak scoping. We wish for the output *c* to be ignored by the consumer, which should stutter when the *BufferWithCounter* takes the transition that produces *c*. When *BufferWithCounter* is composed with *Consumer*, the composition should have an output *c*.

An alternative way to show that the output of one automaton is destined for another is shown in Fig. 12, which shows a composition of *BufferWithCounter* (Fig. 11) with *Consumer* (Fig. 2a). When forming this composition, we wish to ignore all inputs and outputs except those shared by the two automata. In Fig. 12, that sharing is indicated by explicitly connecting the ports rather than by name matching. Moreover, which inputs and outputs are exported to become inputs and outputs of the composition is also shown explicitly. In this example, it is all the inputs and output that are not shared by the two automata. It is now become clear why our visual notation shows the input and output symbols as ports as well as annotations on the transitions. These ports are essential to hierarchical design with strong scoping. Note further that this notation eliminates the need to refer to an output by the same name as that of the corresponding input in another automaton, although we do not exploit that possibility here.

Formally, to "ignore" some inputs or outputs when forming a composition, we simply convert any transitions labeled with these inputs and outputs into internal transitions. We can view this operation as a *projection* of one automaton to the subset of inputs and outputs shared with another automaton. The projection of *BufferWithTransient* and *BufferWithCounter* to *Consumer* are shown in Figs. 13 and 14, 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.

In this case, we can check that *Consumer* is compatible with *BufferWithTransientToConsumer* (Fig. 13), and from this, we can infer without checking that *Consumer* is compatible with *BufferWithCounterToConsumer* (Fig. 14). When using explicit connections as in Fig. 12, we can always tell which subset of the inputs and outputs is pertinent to a composition, and hence how to check compatibility.

The key result is that 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 $P \times R$ and $P' \times R$ 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 $P \times R$ are changed to internal transitions in $P' \times R$. Also, These two product automata have the

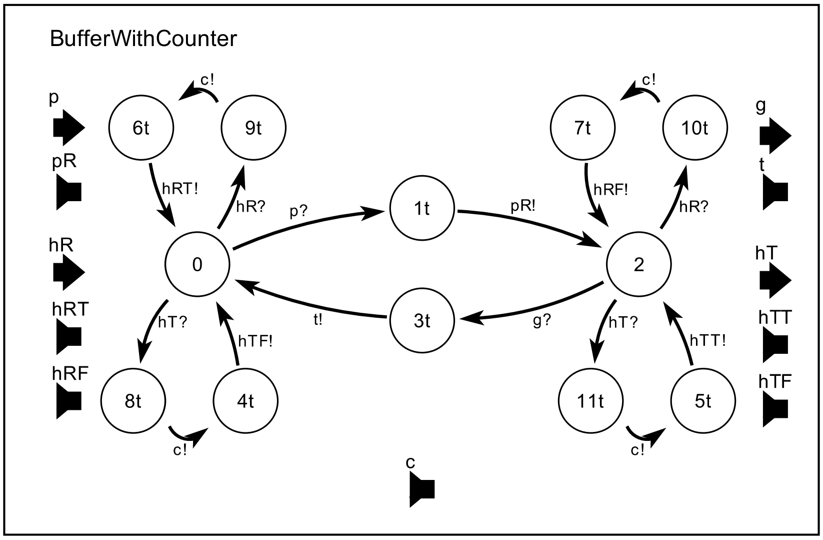

Fig. 11. A buffer that counts the overhead methods

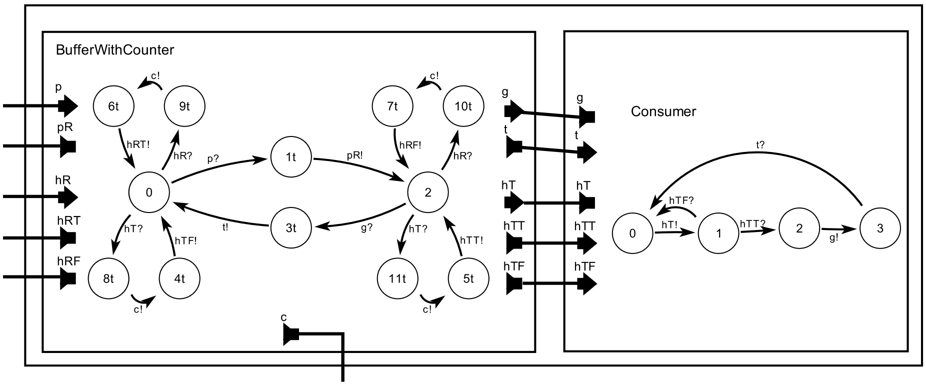

Fig. 12. Explicit connections and hierarchy provide scoping. Here, only the shared inputs and outputs (connected in the middle) are involved in checking compatibility

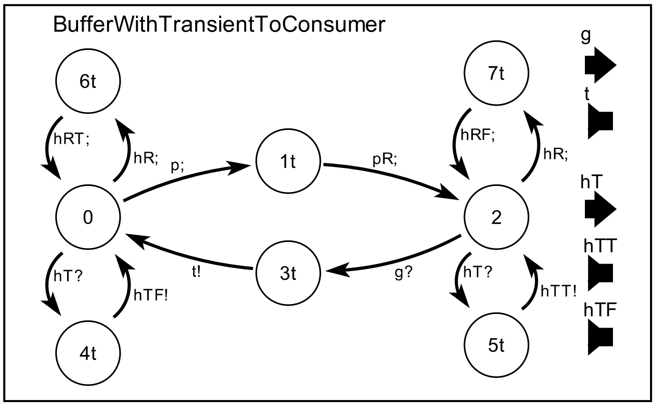

Fig. 13. The projection of *BufferWithTransient* onto *Consumer*

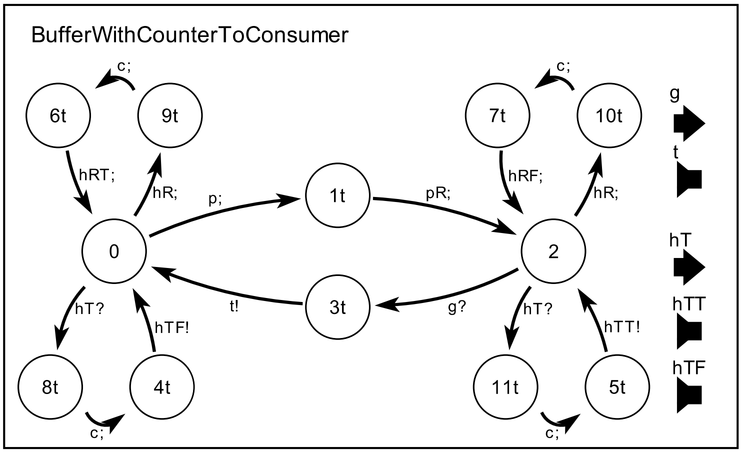

Fig. 14. The projection of *BufferWithCounter* onto *Consumer*

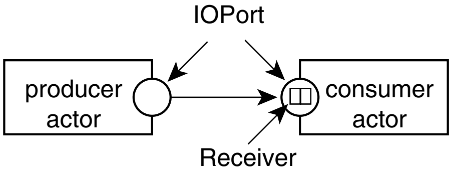

Fig. 15. A simple model in Ptolemy II

same set of illegal states. Furthermore, for all the states in $P \times R$ that can reach illegal states through internal and output transitions, there corresponding states in $P' \times R$ 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 $P' \times R$ is a super set of that of $P \times R$. So if the composition of $P' \times R$ is not empty, $P \times R$ is not empty.

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 Fig. 15. 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:

- *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* or *behaviorally-polymorphic* actors. One of the goals of the behavioral type system is to facilitate the design of behaviorally-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 Fig. 16, the automaton *SDFConsumer* describes a consumer actor designed for the SDF (synchronous dataflow) domain. The inputs and outputs of the automaton are:

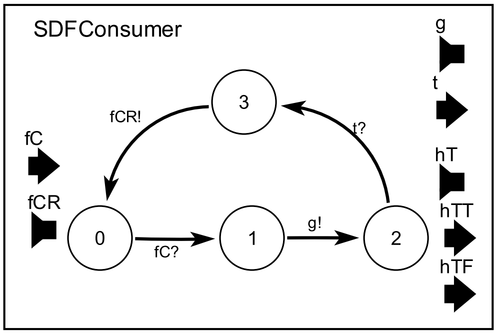

Fig. 16. Interface automata model for an SDF consumer actor

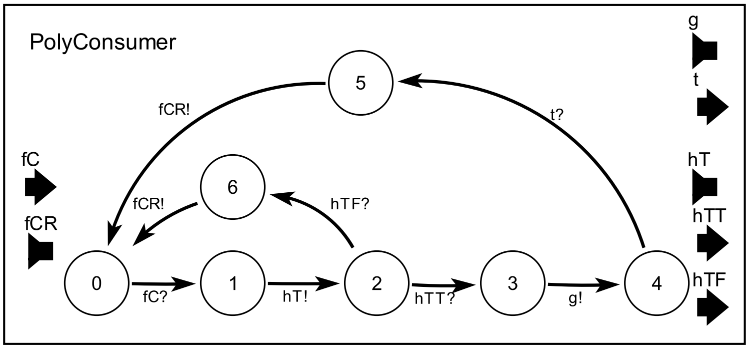

Fig. 17. Interface automaton for a behaviorally-polymorphic consumer actor

- 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 or 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 Fig. 17 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 behavioral polymorphism.

In Ptolemy II, actors interact with the director and the receivers of a domain. In Figs. 16 and 17, 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 Sect. 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 18 shows such an automaton for the SDF domain. Here, p and

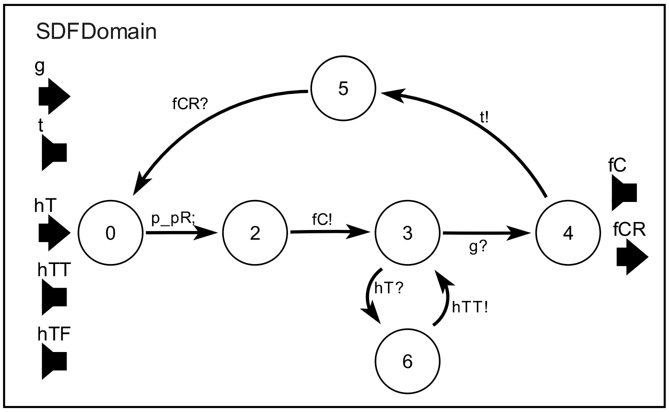

Fig. 18. Type signature of the SDF domain

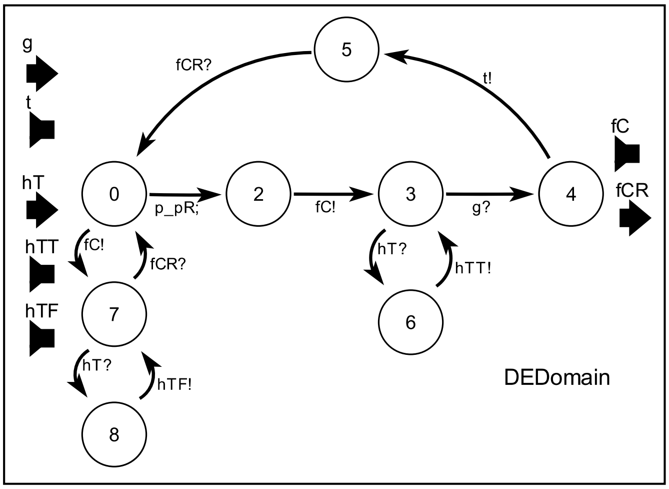

Fig. 19. Type signature of the DE domain

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. 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 Fig. 19. 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 18 and 19 also reflect the fact that both of the 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 20 and 21 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. Conditional rendezvous is implemented by a set of classes in Ptolemy II: *ConditionalBranchController*, *ConditionalSend*, and *ConditionalReceive*. Their operation can also be modeled by interface automata.

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

[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.

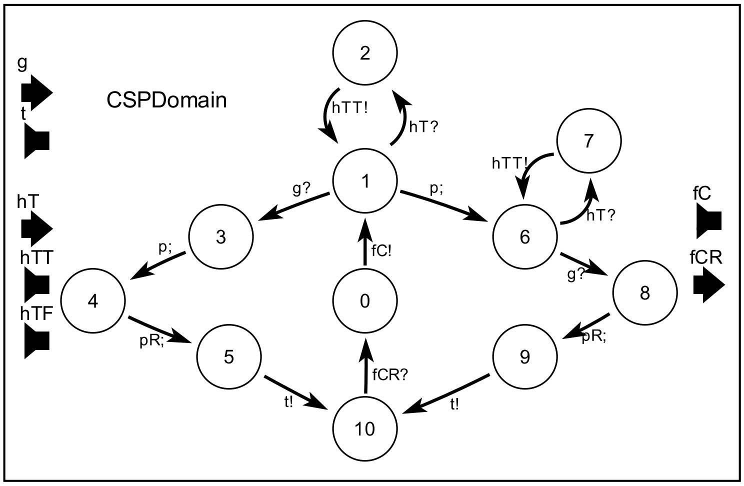

Fig. 20. Type signature of the CSP domain

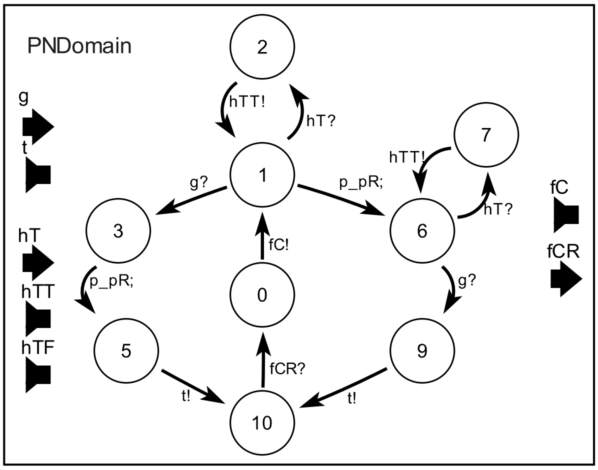

Fig. 21. Type signature of the PN domain

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 Fig. 21, while the case where put() is called first is modeled by the transitions among the states 1, 6, 9, 10.

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 Sect. 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.

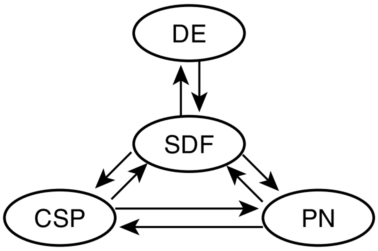

Fig. 22. A directed graph showing the alternating simulation relation among domain types

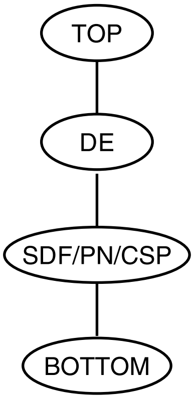

Fig. 23. An example of behavioral-level type order

This is shown by the directed graph in Fig. 22. 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 Sect. 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 behaviorally 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 Fig. 22. However, if we combine the strongly connected components (SCC) into one node, the component graph becomes a partial order. In Fig. 22, there is one SCC consisting of SDF, PN, and CSP. The partial order induced by the component graph is shown in Fig. 23. 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 Fig. 24. 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. They may not be directly useful in practice, but they provide helpful insights. We will discuss these two automata further in Sect. 6.3.

When studying the compatibility with actors, the behavioral type order gives a good description for the relation among various behavioral types. From Fig. 23, 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 Fig. 16 can indeed work in the *SDFDomain*, we compose it with the *SDFDomain* autom-

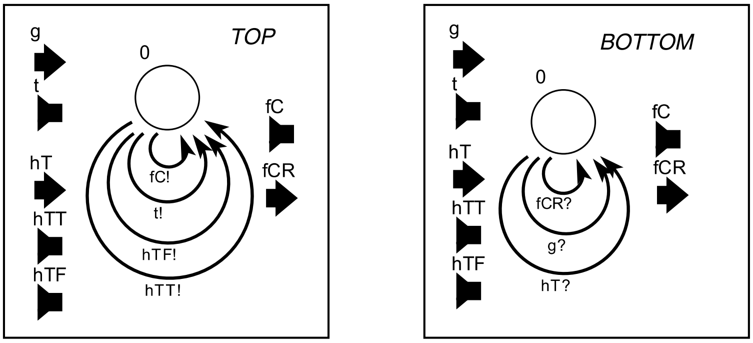

Fig. 24. *TOP* and *BOTTOM* of the behavioral type order

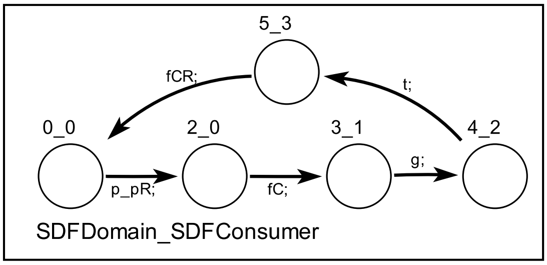

Fig. 25. Composition of *SDFDomain* in Fig. 18 and *SDFConsumer* in Fig. 16

aton in Fig. 18. The result is shown in Fig. 25. As expected, the composition is not empty so *SDFConsumer* is compatible with *SDFDomain*.

Now lets compose *DEDomain* with *SDFConsumer*. The result is an empty automaton shown in Fig. 26. 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 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, *SDFConsumer* 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 Fig. 17 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 Fig. 27. Since *SDFDomain* is below *DEDomain* in the behavioral type order of Fig. 23, we have also verified that *PolyConsumer* can work in the SDF domain. Therefore, *PolyConsumer* is behaviorally polymorphic. As a sanity check, we have composed *SDFDomain* with *PolyConsumer*, and the result is shown in Fig. 28.

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 behaviorally-polymorphic actors. The way that many of these actors consume and process tokens can be modeled by the *PolyConsumer* automaton.

Fig. 26. Composition of *DEDomain* in Fig. 19 and *SDFConsumer* in Fig. 16, which is an empty automaton

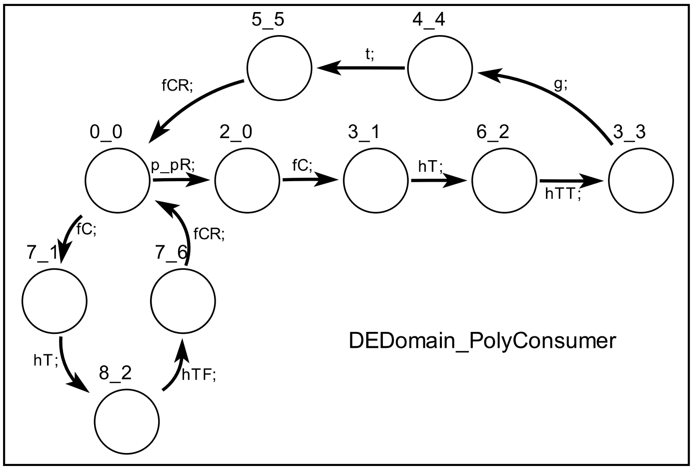

Fig. 27. Composition of *DEDomain* in Fig. 19 and *PolyConsumer* in Fig. 17

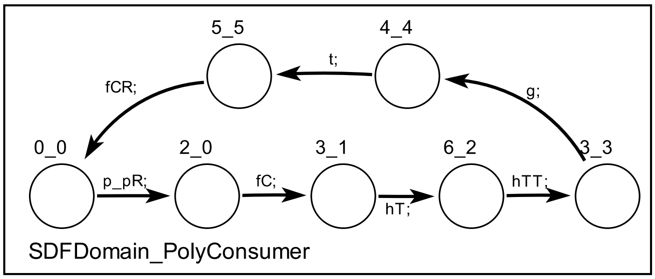

Fig. 28. Composition of *SDFDomain* in Fig. 18 and *PolyConsumer* in Fig. 17

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

Starting from SDF again, Fig. 29 shows an SDF director for the producer/consumer model in Fig. 15. 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 30 shows an SDF receiver. This receiver is more general than the one described in *SDFDomain* in Fig. 18 in that it can hold multiple tokens. In particular, at state 2, where a put() call is just returned, the receiver

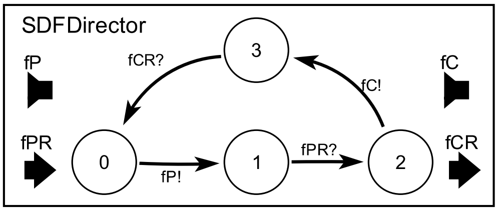

Fig. 29. An SDF director

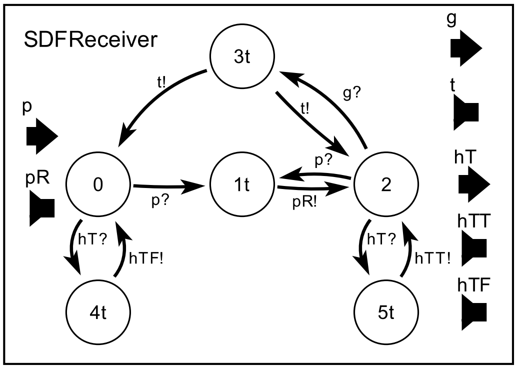

Fig. 30. An SDF receiver

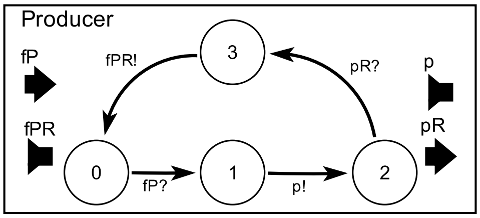

Fig. 31. A producer

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 states 1t, 3t, 4t, and 5t are transient, for reasons similar to the transient states in Fig. 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 31 describes the behavior of a typical producer actor. It simply calls put() in its fire() method. As a type checking example, we can compose *SDFDirector*, *SDFReceiver*, *Producer*, and *SDFConsumer* together, as shown in Fig. 32.

Now let's look at the DE domain. Figures 33 and 34 shows 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 Fig. 35.

For the producer/consumer model, the director in the PN domain will create two threads to run the producer and consumer, as shown in Fig. 36(a), (b). The whole PN director is the composition of these two, as shown in Fig. 36(c). The receiver for the PN domain is shown in Fig. 37. 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 Sect. 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 Fig. 30, the hasToken() call returns *false* at state 4t, while the *PNReceiver* in Fig. 37 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

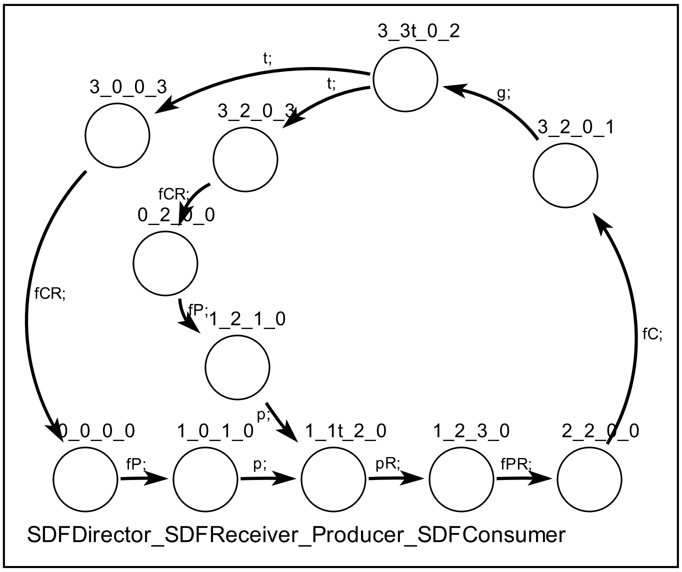

Fig. 32. The composition of *SDFDirector*, *SDFReceiver*, *Producer*, and *SDFConsumer*

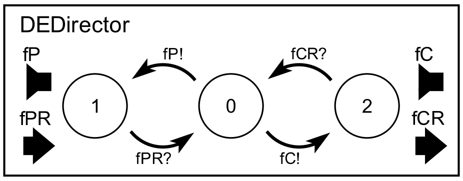

Fig. 33. A DE director

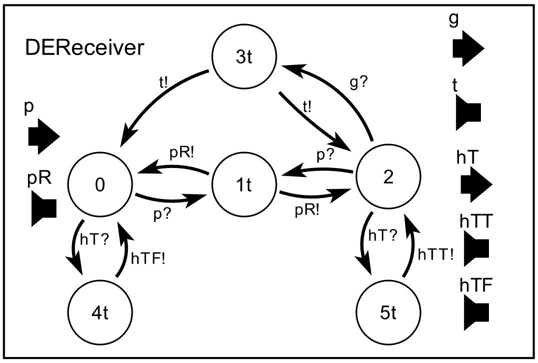

Fig. 34. A DE receiver

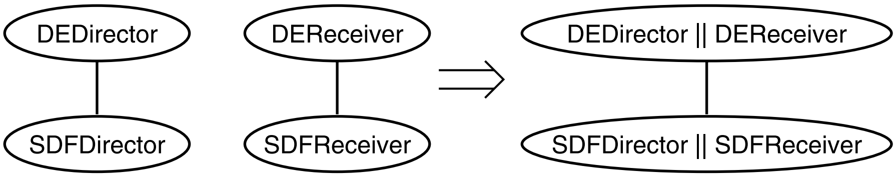

Fig. 35. Using the compositional property to show that *SDFDomain* is a subtype of *DEDomain*

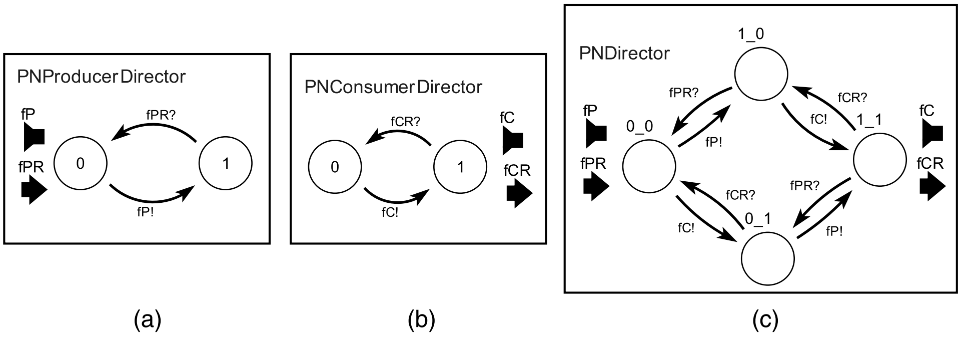

Fig. 36. PN director

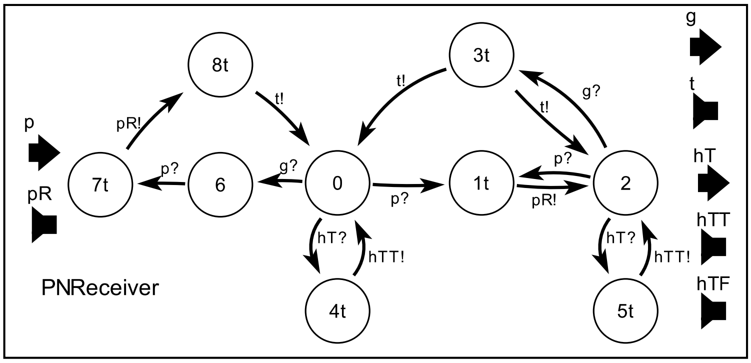

Fig. 37. A PN receiver

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. Fig. 38(a), (b) show the projection of *SDFDirector* and *PNDirector* onto the *Producer* in Fig. 31. We can verify that (1) There is an alternating simulation from *PNDirectorToProducer* to *SDFDirectorToProducer*; (2) *SDFDirectorToProducer* is compatible with *Producer*. So according to the result in Sect. 3.2, we know that *PNDirector* is compatible with

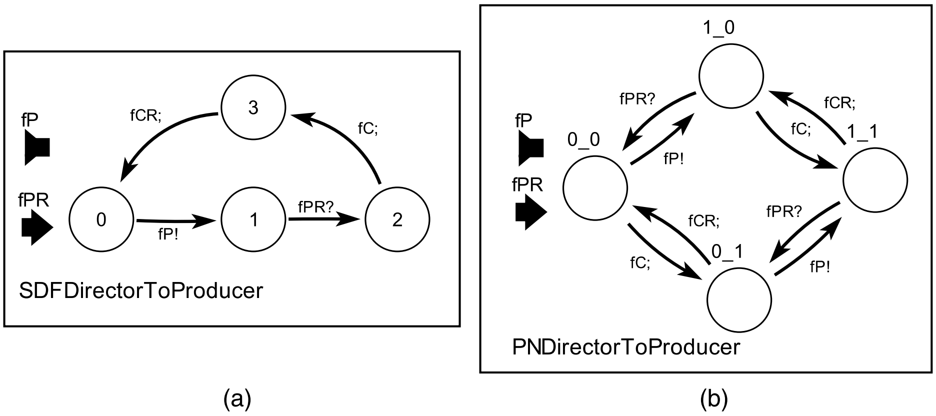

Fig. 38. Projection of *SDFDirector* and *PNDirector* to *Producer*

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 Fig. 36, 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 Fig. 16 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 Sects. 5.1 and 5.4 shows 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 designs. In some cases, such as in Sect. 5.4, we can increase the set of relations captured by using the projection automata, but this is not always possible. So the trade-off between the amount of information carried by the types and the number and structure of subtyping relations in the system also needs to be considered.

Another dimension of the trade-offs is static versus run-time type checking. The examples in the last Sect. 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.

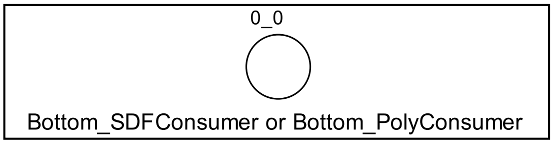

Fig. 39. Composition of *BOTTOM* and *SDFConsumer* or *PolyConsumer*

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 Sect. 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 Fig. 24. These two automata are very general in that they are not only the top and bottom elements of the partial order in Fig. 23, 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 other words, there is an alternating simulation relation from any automaton to the *TOP* automaton in Fig. 24, and an alternating simulation relation from the *BOTTOM* automaton in Fig. 24 to any automaton with the same inputs and outputs.

The *TOP* automaton simply uses the signature of the component's interface and abstracts from all specific behaviors. It does not allow any method calls. 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. The *TOP* automaton may issue any output at any time, and cannot ever accept any inputs, so 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 Fig. 39. The two compositions are the same. Intuitively, since the *BOTTOM* automaton does not have any output transitions, 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 traditional type systems, types abstract the values that an expression may return. Many researchers have proposed extensions to capture the execution behavior for various languages. For example, the effect system [Nie96] augments types with effects, which describe the side-effects that an expression may have, such as read/write effects on the store, or exceptions that may be raised by the expression. One application of effect analysis is in parallel computers. If two expressions do not have interfering effects, then a compiler can schedule them in parallel. In [NiN94], Nielson and Nielson extended the effect system to capture the communication behavior of concurrent ML (CML) programs. In their system, communication behaviors can be included in type inference, and the inference result indicates whether a program only spawns a finite number of processes, or only creates a finite number of channels. In these cases, the compiler may allocate the processes to available processors or allocate communication resources statically.

Behavior analysis has also been applied to actor based languages. For example, Colaco et al. presented a type inference system for a primitive actor calculus [CPS97]. In the actor model, the communication topology is dynamic, so some messages sent out by actors may never be handled. The aim of the inference system is to detect these orphan messages. This system is based on set constraints [Aik94]. It can detect many orphan messages statically and the remaining messages dynamically based on the static type information.

For object-oriented languages, various notions of behavioral subtyping have been proposed [LiW94][LeD00], which extends the conventional OO subtyping to support the specification and checking of invariants and pre and post conditions. For example, an invariant of a bag object is that its size is always less than its bound, and a subtype of bag has to ensure that this invariant holds. While these kinds of properties are important and their violations are the source of many software errors, we are more interested in checking the communication behavior in this work.

Many authors have proposed type systems based on π-calculus or similar calculi. These systems check different properties. For example, Milner's polyadic π-calculus just checks the arities of channels, that is, the number of inputs and outputs along the channel [Mil91]. The extension by Pierce and Sangiorgi captures the polarity, or directionality of channels [PiS93], and Kobayashi, et al. further capture multiplicities, or the number of times the channels can be used [KPT96]. Sumii and Kobayashi proposed a system to ensure deadlock freedom [KSS00]. The type model of Puntigam ensures that all the messages sent by a client process are understood by a server, and Puntigam and Peter later extended it to ensure that the server sends the promised responses [PuP99].

In Ptolemy II, some of these properties are handled by the data-level type system. For example, multiple inputs and outputs in one communication are usually accomplished by passing record tokens and record types are checked together with other data types. Separating the data-level and behavioral-level issues simplifies the type systems. Some other properties, such as channel direction, is naturally encoded in the interface automata representation, and are included in the compatibility check.

Some type systems based on process algebras, such as the behavioral type system of Najm, et al. [NaN99][NNS99] and the system of Ravara and Vasconcelos [RaV97] capture the non-uniform service availability. In both systems, 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. The basic goal here 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, but type checking in interface automata is easier. 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. In [RaV97], a type is defined as a graph with states and transitions, where the transitions represent invoked methods. In these transitions, input, output and internal transitions are not distinguished. We believe the distinction of I/O transitions in interface automata makes them a better choice for modeling software, because there is a natural distinction between the caller and callee of a method, or the sender and receiver of a message. In [NaN99], the definition of subtyping distinguishes the sending and receiving of messages. If a type X2 is a subtype of a type X1, then all the receiving actions of X1 can be performed by X2, and all the sending actions of X2 can be performed by X1. This is analogous to the alternating simulation relation of interface automata and the co/contra-variance relation in function subtyping.

The systems of Igarashi and Kobayashi [IgK01] and Rajamani and Rehof [RaR02] also supports subtyping. The former is a meta type system for the π-calculus, from which a set of type systems checking different properties can be derived. In this system, the subtyping relation $\Gamma_1 \& \Gamma_2 \leqslant \Gamma$ is true, where Γ_1, Γ_2, and Γ are process types, and & represents internal choice. This is not true in our system. In general, if a certain environment Γ is compatible with Γ_1, it may not be compatible with $\Gamma_1 \& \Gamma_2$ since Γ_2 may issue output that is not compatible with Γ. So our definition of subtyping is more appropriate for our purpose. In [RaR02], Rajamani and Rehof used a behavioral type system for checking π-calculus programs. In particular, they checked the conformance of software implementation against a specification. They were interested in checking whether a message-passing software is stuck-free, i.e., a message sent by a sender will not get stuck without received by a receiver, or a receiver waiting for a message will not get stuck without receiving a message from some sender. To be able to check this property at an abstract level, they defined a conformance relation stronger than alternating simulation. In Ptolemy II, stuck-free is not an important property to check, because it is in general not a failure condition. It could be normal for an actor to try to read a token, but not getting it.

Extended type systems for checking behavioral properties have also been developed for popular languages such as C [STS01] and Java [LBR98][LNS00]. However, these systems do not check the compatibility of communication protocols.

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. We do not insert converters in the middle. 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 presented the use of interface automata to devise a behavioral type system for Ptolemy II, a software framework for concurrent component composition. We have leveraged the contravariant and optimistic properties of interface automata to achieve behavioral subtyping and polymorphism, and we have extended interface automata in two ways to deal with atomicity in concurrent actions and with scoping problems. Transient states allow us to model mutual exclusion easily, thus improving the modeling of atomic actions. Projection automata improve the scoping of named inputs and outputs.

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 behaviorally 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 that 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. We also thank the anonymous reviewers for providing critical comments.

This research is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), the Defense Advanced Research Projects Agency (DARPA), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from the State of California MICRO program, and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

References

[Agh86] Agha GA (1986) Actors: a model of concurrent computation in distributed systems, The MIT Press

[Aik94] Aiken A (1994) Set Constraints: results, applications and future directions, In: Proceedings of the second workshop on the principles and practice of constraint programming, Orcas Island, Washington

[AlG97] Allen R, Garlan D (1997) A formal basis for architectural connection, ACM Trans Software Eng. Methodology 6(3)

[BCD02] Bhattacharyya SS, Cheong E, Davis J II, Goel M, Hylands C, Kienhuis B, Lee EA, Liu J, Liu X, Muliadi L, Neuendorffer S, Reekie J, Smyth N, Tsay J, Vogel B, Williams W, Xiong Y, Zheng H (2002) Heterogeneous concurrent modeling and design in java, technical memorandum UCB/ERL M02/23, EECS, University of California, Berkeley (http://ptolemy.eecs.berkeley.edu/publications/papers/02/ptIIdesign/)

[BLO98] Borriello G, Lavagno L, Ortega RB (1998) Interface synthesis: a vertical slice from digital logic to software components, In: Proceedings of international conference on computer aided design (ICCAD), San Jose

[CAR97] Cardelli L (1997) Type Systems, The computer science and engineering handbook, CRC Press

[CaW85] Cardelli L, Wegner P (1985) On understanding types, data abstraction, and polymorphism, ACM computing surveys 17(4)

[COB92] Chou P, Ortega RB, Borriello G (1992) Synthesis of the hardware/software interface in microcontroller-based systems, Proceedings of ICCAD, pp 488–495

[COB95] Chou P, Ortega RB, Borriello G (1995) Interface co-synthesis techniques for embedded systems, In: Proceedings of the international conference on computer aided design

[CPS97] Colaco, J-L, Pantel M, Salle P (1997) A set-constraint-based analysis of actors, In: International workshop on formal methods for open object-based distributed systems, Canterbur, UK

[DaP90] Davey BA, Priestly HA (1990) Introduction to lattices and order, Cambridge University Press

[deH01] de Alfaro L, Henzinger TA (2001) Interface Automata, In: Proceedings of the joint 8th European software engineering conference and 9th ACM SIGSOFT international symposium on the foundations of software engineering (ESEC/FSE 01), Austria

[deH01a] de Alfaro L, Henzinger TA (2001) Interface theories for component-based design, In: Proceedings of the first international workshop on embedded software (EMSOFT), Lecture notes in computer science, vol 2211. Springer, Berlin Heidelberg New York, pp 148–165

[EiP00] Eisenring M, Platzner M (2000) Synthesis of interfaces and communication in reconfigurable embedded systems, IEE Proc Comput Digit Tech 147(3)

[ETT98] Eisenring M, Teich J, Thiele L (1998) Rapid prototyping of dataflow programs on hardware/software architectures, In: Proceedings 31st annual hawaii international conference on system sciences

[FlF00] Flanagan C, Freund SN (2000) Typed-Based Race Detection for Java, In: ACM SIGPLAN conference on programming language design and implementation, PLDI 2000, Vancouver

[GoS02] Goessler G, Sangiovanni-Vincentelli A (2002) Compositional Modeling in Metropolis, In: Proceedings of EMSOFT, Grenoble, France, LNCS. Springer, Berlin Heidelberg New York

[Hoa78] Hoare CAR (1978) Communicating sequential processes, Commun ACM 28(8)

[IgK01] Igarashi A, Kobayashi N (2001) A generic type system for the pi-calculus, In: 28th annual ACM SIGPLAN - SIGACT symposium on principles of programming languages (POPL 2001)

[IWY00] Inverardi P, Wolf AL, Yankelevich D (2000) Static checking of system behaviors using derived component assumptions, ACM Trans Software Eng. Methodol 9(3)

[Kah74] Kahn G (1974) The semantics of a simple language for parallel programming, In: Proceedings of the IFIP congress 74, North-Holland Publishing Co.

[KaM77] Kahn G, MacQueen DB (1977) Coroutines and networks of parallel processes, In: Gilchrist B (ed) Information processing, vol 77. North-Holland Publishing Co.

[KPT96] Kobayashi N, Pierce BC, Turner DN (1996) Linearity and the Pi-Calculus, ACM symposium on principles of programming languages

[KSS00] Kobayashi N, Saito S, Sumii E (1877) An Implicitly-typed deadlock-free process calculus, In: 11th international conference on concurrency theory, CONCUR 2000, Pennsylvania, LNCS 1877

[LeD00] Leavens GT, Dhara KK (2000) Concepts of behavioral subtyping and a sketch of their extension to component-based systems, In: Foundations of component-based systems, Cambridge University Press, pp 113–135

[LBR98] Leavens GT, Baker AL, Ruby C (1998) Preliminary design of JML: a behavioral interface specification language for java, Dept. of Computer Science, Technical Report no. 98–06, Iowa State University, Ames, Iowa 50011

[Lee02] Lee EA (2002) Embedded Software, Advances in computers Zelkowitz M (ed) vol 56. Academic Press, London

[LeM87] Lee EA, Messerschmitt DG (1987) Synchronous Data Flow, Proceedings of the IEEE, 75(9)

[LeV01] Lee EA, Varaiya P (2003) Structure and interpretation of signals and systems, Addison-Wesley

[LeX01] Lee EA, Xiong Y (2001) System-level types for component-based design, First workshop on embedded software, EMSOFT2001, Lake Tahoe, CA, USA

[LNS00] Rustan K, Leino M, Nelson G, Saxe JB (2000) ESC/Java User's Manual, Technical Note 2000-002, Compaq Systems Research Center

[LiW94] Liskov BH, Wing JM (1994) A behavioral notion of subtyping, ACM Trans Programming Lang Syst 16(6)

[LyT81] Lynch N, Tuttle M (1981) Hierarchical correctness proofs for distributed algorithms, In: Proceedings 6th ACM symposium principles of distributed computing, pp 137–151

[Mil91] Milner R (1991) The Polyadic π-calculus: a Tutorial, Technical Report ECS-LFCS-91-180, Department of computer science, University of Edinburgh, UK, Reprinted in Bauer FL, Brauer W, Schwichtenberg H (ed) Logic and algebra of specification, Springer, Berlin Heidelberg New York, 1993

[MPW92] Milner R, Parrow J, Walker D (1992) A calculus of mobile processes (Part I and part II), Inf Comput 100:1–77

[Mit84] Mitchell JC (1984) Coercion and type inference, In: 11th annual ACM symposium on principles of programming languages, pp 175–185

[NaN99] Najm E, Nimour A (1999) Explicit behavioral typing for object interface, In: Semantics of objects as processes, ECOOP'99 Workshop, Lisbon, Portugal

[NNS99] Najm E, Nimour A, Stefani J-B (1999) Infinite types for distributed object interfaces, In: Third IFIP conference on formal methods for open object-based distributed systems (FMOODS'99), Firenze, Italy

[Nie96] Nielson F (1996) Annotated type and effect systems, ACM Comput Surveys 28(2)

[NiN94] Nielson HR, Nielson F (1994) Higher-order concurrent programs with finite communication topology, ACM Symposium on principles of programming languages

[Ode96] Odersky M (1996) Challenges in type system research, ACM Comput Surveys 28(4)

[OrB97] Ortega RB, Borriello G (1997) Communication synthesis for embedded systems with global considerations, In: Proceedings of the 5th international workshop on hardware/software co-design (Codes/CASHE'97)

[PRS98] Passerone R, Rowson JA, Sangiovanni-Vincentelli A (1998) Automatic synthesis of interfaces between incompatible protocols, 35th design automation conference

[PiS93] Pierce B, Sangiorgi D (1993) Typing and subtyping for mobile processes, In: Logic comput sci

[Pun96] Puntigam F (1996) Types for active objects based on trace semantics, In: Proceedings of the workshop on formal methods for open object-oriented distributed systems (FMOODS'96), France

[PuP99] Puntigam F, Peter C (1999) Changeable interfaces and promised messages for concurrent components, Proceedings of the ACM symposium on applied computing (SAC'99), San Antonio, Texas

[RaR02] Rajamani SK, Rehof J (2002) Conformance checking for models of asynchronous message passing software, Conference on computer aided verification, CAV2002, Copenhagen, LNCS 2404

[RaV97] Ravara A, Vasconcelos VT (1997) Behavioural types for a calculus of concurrent objects, In: 3rd international euro-par conference, LNCS 1300, pp 554–561

[STS01] Shankar U, Talwar K, Foster JS, Wagner D (2001) Detecting format-string vulnerabilities with type qualifiers, 10th USENIX security symposium

[XiP98] Xi H, Pfenning F (1998) Eliminating array bound checking through dependent types, In: Proceedings of ACM SIGPLAN Conference on programming language design and implementation (PLDI '98), Montreal pp 249–257

[Xio02] Xiong Y (2002) An extensible type system for component-based design, PhD Thesis, Technical memorandum, UCB/ERL M02/13, University of California, Berkeley

[XiL00] Xiong Y, Lee EA (2000) An extensible type system for component-based design, In: 6th international conference on tools and algorithms for the construction and analysis of systems, LNCS 1785, Berlin

Received September 2002
Accepted in revised form November 2003 by M. Broy, G. Lüttgen and M. Mendler
Published online 11 May 2004