# Counting Interface Automata and their Application in Static Analysis of Actor Models

Ernesto Wandeler* Jörn W. Janneck† Edward A. Lee† Lothar Thiele*

## Abstract

*We present an interface theory based approach to static analysis of actor models. We first introduce a new interface theory, which is based on Interface Automata, and which is capable of counting with numbers. Using this new interface theory, we can capture temporal and quantitative aspects of an actor interface as well as an actor's token exchange rate. We will show, how to extract this information from actors written in the Cal Actor Language (CAL), and we also present a method to capture the interface information as well as the structure of dataflow models into an interface automaton. This automaton acts as glue between the automata of all actors in the model, and by successfully composing all actor automata with it, we can prove interface compatibility of all actors with the composition framework. After successful composition, the resulting automaton will contain information that can be used for further static analysis of the composite actor model.*

## 1 Introduction

Component based design is an approach to software and system engineering in which new software designs are created by combining pre-existing software components. The ability to check the compatibility of such components, and to detect errors in the composite system of components is thereby an important factor for software development productivity and software quality.

Many modern programming languages which make extensive use of component libraries, such as Java or C#, provide a component interface compatibility checking on a data type level. By checking whether the data types of arguments in a method call to a component are compatible with the specified interface they ensure a certain level of component compatibility.

Interface theories for component based design [6] take this interface compatibility checking one step further by providing formalisms to specify component interfaces in more detail. One recent approach in this area is Interface Automata [5], which is an interface theory defining a lightweight formalism that can capture the temporal aspects of software component interfaces. In particular, Interface Automata can capture assumptions on the order in which the methods of a component may be called, as well as guarantees on the order in which a component itself calls external methods.

In the Ptolemy II project [12, 2], which studies actor based modeling, simulation and design of concurrent, real-time, embedded systems, efforts were made recently to use Interface Automata to capture the temporal aspects of an actor's interface towards its composition framework, into what was called the *behavioral type* of an actor [15]. Using this behavioral type, type checking can be done towards the behavioral type of an actor composition framework, to ensure compatibility on a behavioral type level between an actor and its composition framework.

**Contributions** In this paper, we present a new interface theory that is tailored towards the use in actor systems and that allows us to capture more aspects of an actor's interface into its behavioral type than we could with original Interface Automata. We call this interface theory Counting Interface Automata (CIA).

We present a method to extract the interface information of an actor written in the Cal Actor Language (CAL) [8, 1] into a CIA, and we also present how to create a CIA of a composition framework with a dataflow model of computation.

With the successful composition of this framework automaton and all actor automata, we can prove behavioral type compatibility of all actors with the composition framework.

The successful composition leads to a new CIA, which contains the complete token exchange information of the

[1]Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH), 8092 Zürich, Switzerland.
E-mail:{wandeler, thiele}@tik.ee.ethz.ch
[2]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 94720, USA.
E-mail:{janneck, eal}@eecs.berkeley.edu

composite actor model and which can be used for further static analysis of the composite actor model. We will show, how the token exchange information contained in this automaton can be extracted.

## 2 An Introduction to Actor Based Modeling

Actor based modeling is an approach to systems design in which entities called actors [10, 3] communicate with each other through ports and communication channels. From the point of view of component based design, actors are the components in actor oriented modeling.

In the context of this work, an *actor* is a computational entity with a well defined component interface. It has *input ports*, *output ports*, *state* and *parameters*. An actor communicates with other actors by sending and receiving atomic pieces of data, called *tokens*, through its ports, along unidirectional connections, called *channels*. Actors are connected through channels to form *models*. When an actor is executed it is said to be *fired* and it may consume tokens from its input ports, produce tokens on its output ports and update its internal state, based on consumed tokens, its parameters and its former state.

The syntactic structure of an actor-oriented design alone says little about the semantics of the model. The semantics is largely orthogonal to the syntax and is determined by a so-called model of computation (MoC). A MoC governs the interaction of actors in a model by defining operational rules for executing the actors as well as for the communication between actors.

In this work we concentrate on two dataflow models of computation. In dataflow models, actor computations are triggered by the availability of input data, and connections between actors represent the flow of data from a producer actor to a consumer actor and are typically buffered with FIFO queues. Dataflow models are especially useful to model data-driven processing as in signal processing. The two different dataflow models of computation we use in this work are:

**SDF** - Synchronous Dataflow: The synchronous dataflow model of computation [13] is a particularly restricted special case of dataflow, where all actors in a model must have constant data consumption and production rates. This leads to the extremely useful property that deadlock and buffer boundedness are decidable, and moreover, the scheduling of actor firings as well as the buffer capacity for communication can be computed. Note, that SDF in this context is not the SDF of Lustre [9], although it may be possible to adapt these methods to the SDF of Lustre.

**DDF** - Dynamic Dataflow: The dynamic dataflow model of computation does not restrict the data consumption and production rates of actors, but properties such as deadlock and boundedness are often not statically decidable.

We will call the actor model framework in which an actor is embedded its composition framework. The composition framework thereby contains all information of the actor model except the actors themselves, i.e. it contains the model of computation together with firing strategies for the firing of the actors, as well as the actor's interconnection information in the model.

For communication, the composition framework provides a set of primitive communication operations that allow an actor to query the state of communication channels and to retrieve or send information from and to channels:

- *get(k)* retrieves *k* data tokens via a port,
- *put(k)* produces *k* data tokens via a port,
- *hasToken(k)* tests whether *get(k)* can be successfully called on a port,
- *hasRoom(k)* tests whether *put(k)* can be successfully called on a port.

These communication operations might be implemented differently, depending on the model of computation, but they always must be present and actors should only communicate using these operations [14]. The exact way an actor is using these operations, determines whether it can work under a given model of computation in composition framework, and is captured into what we call an actor's behavioral type [15].

## 3 An Automata Based Analysis Strategy

The ability to analyze systems in component based design, in order to check the compatibility of components and to detect errors in the composite system, is an important factor for system development productivity and system quality.

Instead of performing such analysis directly on the source code of complete actor models, we propose to use a light-weight formalism, based on interface automata, to capture only selected aspects of the actors and the composition framework, which are important for the analysis.

Figure 1 depicts this analysis strategy. On the top, an actor model and its composite actor are shown. The transformation $c_{MoC}$ is a program transformation from an actor model with a particular model of computation into its composition [11]. To analyze such a composite actor from its source code is usually very hard, in particular if it is implemented using a conventional Turing-complete imperative programming language. This is because then much of

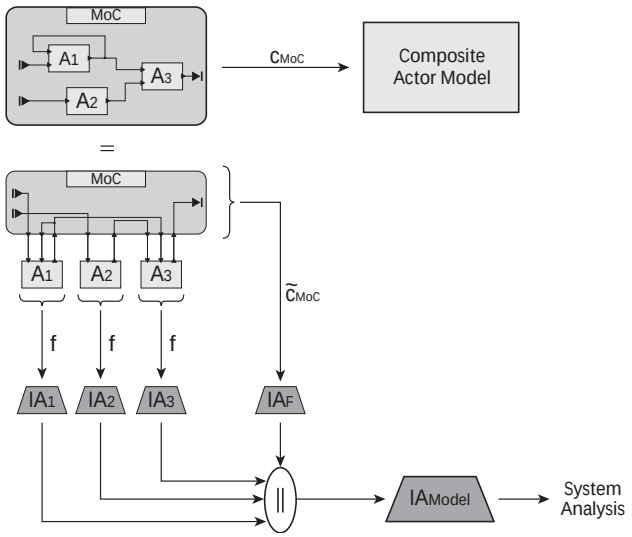


**Figure 1. The relation of an actor model, its composition and interface automata.**

the important information is often not easily accessible anymore or is even not contained explicitly anymore.

In our analysis strategy, we independently extract the necessary information of all actors and the composition framework into separate interface automata at the very beginning, and we then build the composite actor model in the interface automata domain, where the necessary information is preserved during composition.

The framework automaton $IA_F$ is extracted from the actor model using a transformation $\tilde{c}_{MoC}$. It contains all information needed from the composition framework, namely the behavioral type of the model of computation, the interconnection information of the actors in the model and the firing schedule of the actor model.

The actor automata $IA_1$, $IA_2$ and $IA_3$ on the other hand are extracted from the actors $A_1$, $A_2$ and $A_3$, using a transformation $f$, and contain the behavioral type of the respective actor in the actor model together with its token consumption and production rate.

This clear separation of actors and their composition frameworks in the automaton domain reflects the original idea of actor oriented modeling and enables effortlessly replacing one actor automaton by another actor automaton in the automata domain, as we would replace one actor by another actor in the real actor model.

After building these automata, the successful composition of all actor automata with the framework automaton corresponds to behavioral type checking and ensures component compatibility of all actors with the model of computation of the actor model. The result of this composition is an interface automaton $IA_{Model}$ of the composite actor model, which explicitly contains the interconnection information of the actors, and the firing schedule as well as the complete internal token exchange rates of the composite actor model. This information can be used for further static analysis of the actor model to detect errors in the composite system.

In order to capture all the above described information of actors and their composition frameworks, we present a new interface theory which is tailored towards the use in actor systems and which we call Counting Interface Automata (CIA).

## 4 Counting Interface Automata

In this section, we provide an informal description of Counting Interface Automata. For a strictly formal definition, which we cannot give here due to space restrictions, please refer to [18]. In Section 4.3 we will discuss some aspects of Counting Interface Automata and highlight differences to other interface theories as Interface Automata [5], Resource Interfaces [4] and Timed Interfaces [7].

### 4.1 Automaton

Counting interface automata consist of a finite set of *states*, of which one is the *initial state*, and a finite set of *transitions*. A subset of the states may be labeled as *reset states*. As other automata, Counting Interface Automata are usually depicted by bubble-and-arc diagrams.

We distinguish three different kinds of transitions, namely *input transitions*, *output transitions* 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 modeled component. Internal transitions correspond to computations inside the component.

All transitions are labeled by *actions*, which are also divided into *input actions*, *output actions* and *internal actions*, depending on the kind of transition they label.

As the name suggests, Counting Interface Automata have the ability to count with numbers. To keep track of counted values, the automaton has a finite set of *counter variables*, which can be divided into two disjoint sets of so-called *persistent counter variables* and *transient counter variables*. The values of these counter variables may represent anything countable of interest, as for example the length of a token queue in a communication channel of an actor model. The difference between the persistent counter variables and the transient counter variables is that the latter ones are reset to the value *nil* in every reset state, while the

others keep their value. We will see the importance of this distinction in Section 6.

Every transition may contain a *guard*, which expresses a condition on the counter variable values, and which determines whether the transition is enabled. Furthermore, a transition may contain a set of *counter declarations* and *counter assignments*, which declare new counter variables or update their values respectively when the transition is taken.

So called *action quantities* on input and output transitions express quantitative aspects of actions on a transition and may also be used to exchange and compare counter variable values between two automata during their composition. When modeling a software component, an action quantity could correspond to a countable argument of a method call, as for example the number of data tokens that get consumed during the method call. Action quantities are particularly interesting during composition, where the equality of quantities is an additional constraint on synchronized shared transitions.

In place of an action quantity, an input transition may also have a so called *action quantity declaration*. In this case, the action quantity of the input transition does not express a restriction but is instead open and may take on any action quantity value during composition with another automaton. This value then gets bound to a counter variable which is specified in the action quantity declaration. This mechanisms allows two automata to exchange counter values during composition and is one of the main features of Counting Interface Automata.

For the semantics of the automaton, it is important to note that on a transition, the guard is evaluated after a possible action quantity declaration, but before all counter declarations and counter assignments.

**Example 1** *Figure 2 shows an automaton describing the behavioral type of a DDF model of computation towards an actor with one input port and one output port, and its output port connected to two input ports of other actors. The number of tokens available in the buffer of the input ports of the actor and the two connected actors are represented by the persistent counter variables T, Ta and Tb respectively.*

*State 0 is the initial state and an internal action* step *triggers the start of an actor firing. Then in state 1, the model of computation fires an actor, expressed by the output action* f. *In state 2, the model of computation is ready to receive any input action* hT, g, hR, p *or* fR, *corresponding to an actors method calls* hasToken(n), get(n), hasRoom(n), put(n) *or the return of fire.*

*All input transitions leaving from state 2, except* fR, *have an action quantity declaration, expressing that the calling actor can pass an argument, namely its own action quantity, which is then bound to the transient counter variable* n.

*Using guards and the transient counter variable* cT, *which represents the number of tokens whose availability was checked using the* hasToken(n) *method, the automaton can ensure that the actor can only call the* get(n) *method after checking the availability of at least* n *tokens. During a call to* get(n), *both counter variables* T *and* cT *get decremented by* n. *Further, a call to* hasRoom *always returns true and a call to* put(n) *increases the counter variables* Ta *and* Tb, *which represent the input port buffers of the two connected actors, by* n.

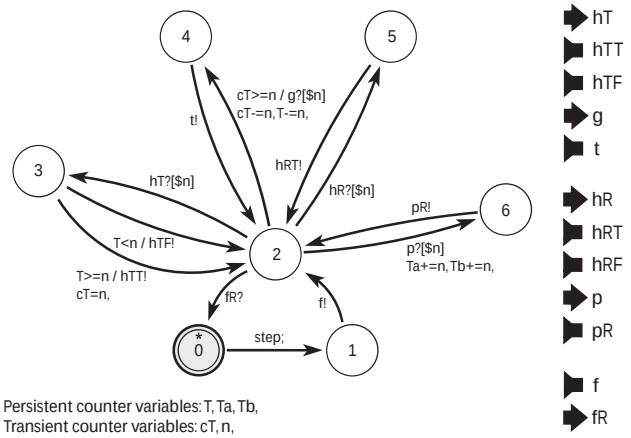


**Figure 2. An automaton describing the behavioral type of a DDF model of computation. Input, output and internal actions are labeled with "?", "!" and ";" respectively and reset states are grey colored. On the right side, input and output actions of the automaton are listed, depicted as communication ports of the automaton to other automata. Note that action quantities equal to 1 are often not shown explicitly.**

When describing a software component, a path through a Counting Interface Automaton corresponds to an execution of a part of the software component. Therefore, we call a state trajectory an *execution fragment*. We call an execution fragment which starts at the initial state of an automaton an *execution path* of the automaton. Further, an execution fragment or execution path is called *unconditional*, if it contains only output and internal transitions. Otherwise an execution fragment is called *conditional*. This distinction is important, because the execution of a conditional execution fragment can be avoided by the environment, while the execution of an unconditional execution fragment cannot be avoided.

### 4.2 Composition

Two Counting Interface Automata are only *composable*, if their sets of input actions, output actions and counter variables are disjoint respectively. Input actions of one automaton, may however coincide with output actions of the other automaton. These actions are then called *shared actions* of the two automata.

To get the composition of two composable automata, we first build their *product*. The product of two automata consists of the union of their counter variables and the product of their states. The transitions of the original automata which are labeled with non-shared actions are still present in the product automata, while pairs of transitions with shared actions get synchronized and become a single transition with an internal action in the product automaton. A *state invariant*, which is added to the start state of such a shared synchronized transition during product building, ensures that either the guard of the shared output transition must evaluate to false or that otherwise the guard of the shared input transition must evaluate to true and the action quantities of the shared action must match. Recall that a shared input action with an action quantity declaration matches with any output action quantity.

Further, if an output transition labeled with a shared action starts at a state in the product where no corresponding input transition labeled with the same shared action exists, another state invariant must ensure that the guard of the output transition never evaluates to true.

Note, since guards on a transition are evaluated after a possible action quantity declaration, the counter context with which state invariants are evaluated is the one after evaluating all possible action quantity declarations of leaving transitions.

A state in the product automaton is an *error state* if for any unconditional execution path or for all conditional execution paths which lead to this state, the state invariant evaluates to false.

We now get the *composition* of the two initial automata by removing all states in the product from where there exists an unconditional execution fragment to any error state. If this composition is not empty, the two initial automata are said to be *compatible*.

An example of the composition of two Counting Interface Automata is given in Example 4.

### 4.3 Discussion

The composition and with it the question of compatibility of two Counting Interface Automata is undecidable in general. In Section 6 we will however see that generated actor and composition framework automata exhibit special topological structures that make the question of composition and compatibility decidable for behavioral type checking in composite actor models.

Compared to original Interface Automata [5], Counting Interface Automata allow us to create more refined interface descriptions of both actors and composition frameworks.

Counting Interface Automata let us express the token consumption and production rate of an actor in its interface. Using original Interface Automata, we could express this information using a number of consecutive calls to *getToken()* or *putToken()*, but the behavioral type of the composition framework would then be required to be ready to receive exactly the same number of calls to these methods. This would break the separation of actors and their composition framework that is central to actor based modelling. Alternatively, the composition framework could be ready to receive any number of calls to these methods, in which case the behavioral type of the composition framework could not express anymore that an actor may only consume the number of tokens whose availability he initially checked using a call to *hasToken()*.

Using the transient counters and guards of Counting Interface Automata, we can express that a composition framework may be ready to receive calls to a number of methods at a time while still putting requirements on partial orders between these methods calls. For example a framework may express that for any call to *getToken(n)* there must have been a call to *hasToken(n)* somewhen in the past, but that there may be any number of calls to *hasRoom()* or *putToken()* in between. Using Interface Automata we could only express this by enumerating all possible valid method call sequences, which would usually be prohibitive.

As a last point to mention, using assignments to persistent counters allows us to model the actor interconnection information of a composition framework and the token exchange information of a complete actor model, which is not possible with Interface Automata.

Two other recent interface theories, Timed Interfaces [7] and Resource Interfaces [4], also have capabilities to count in a certain way. One main difference between them and Counting Interface Automata is the ability of Counting Interface Automata to exchange counted values between automata during composition, using action quantity declarations. The use of this feature is central in the creation of a composition framework automaton as we will see in the next section. Further, while Resource Interfaces do not allow arbitrary counter variables at all, the counter variables in Timed Automata have different semantics than the ones in Counting Interface Automata in that they model continuously increasing time and their values may only increase in automata states.

## 5 Generating Counting Interface Automata

The information needed for an actor's or a framework's counting interface automaton is contained in the actor's source code and the composition framework itself. In this section, we will show how to extract this information from actors written in the Cal Actor Language (CAL) [8, 1] and from frameworks with an SDF or DDF model of computation. Similar methods could be used to extract this information from other actor definition languages as for example StreamIT [17].

### 5.1 Generating Actor Automata

CAL is a domain specific language for defining the functionality and behavior of actors. Its key goal is to make actor programming easier, and to enable an actor programmer to explicitly express the information and behavioral properties of an actor that are relevant to its usage. This facilitates automatic extraction of actor properties needed to generate counting interface automata.

**Example 2** *The code below shows a non-deterministic merge actor, written in* CAL.

```
1: actor Mrg () T In1, T In2 ==> T Out;
2:    A1:action [a], [] ==> [a] end
3:    A2:action [], [a] ==> [a] end
4:    selector
5:       (A1|A2)*
6:    end
7: end
```

*The first line defines the interface of the actor to its environment. This actor has two input ports* In1 *and* In2*, and one output port* Out*, and all require tokens of type* T*. On lines 2 and 3, two actions* A1 *and* A2 *are specified. Both read one token from input port* In1 *or* In2 *respectively, and write the read token out to* Out*. On line 5, an action selector specifies the order in which actions are chosen when the actor is fired, using a finite state machine or equivalently a regular expression. If the actor is fired, one of the actions is chosen and executed, depending on the action selector and the availability of tokens on the input ports.*

The first step of generating the counting interface automaton of a CAL actor, is to transform the actor code from the rather declarative CAL to the imperative Calflow [18], which is an intermediate format of CAL. There, all data dependencies are explicitly resolved and a sequential schedule for so-called *atomic steps* that get executed during the firing of an action, is given. Examples for such atomic steps are checking the availability of tokens, consuming tokens, producing tokens, or computing statements.

With the information on the number of consumed and produced tokens, each of these atomic steps can directly be transformed to a short state trajectory that is specific for every type of atomic step. Put together in the order of the sequential schedule of the original atomic steps, all these short state trajectories lead to a larger state trajectory which describes the interaction of an actor with its composition framework during the execution of one action of the actor.

We then take the action selector in its finite state machine form, and replace all states with reset states of counting interface automata and all transitions, each corresponding to the execution of one action, with the above state trajectories. This leads to the complete counting interface automaton of the actor.

**Example 3** *Figure 3 shows the counting interface automaton of the non-deterministic merge actor* Mrg *from Example 2.*

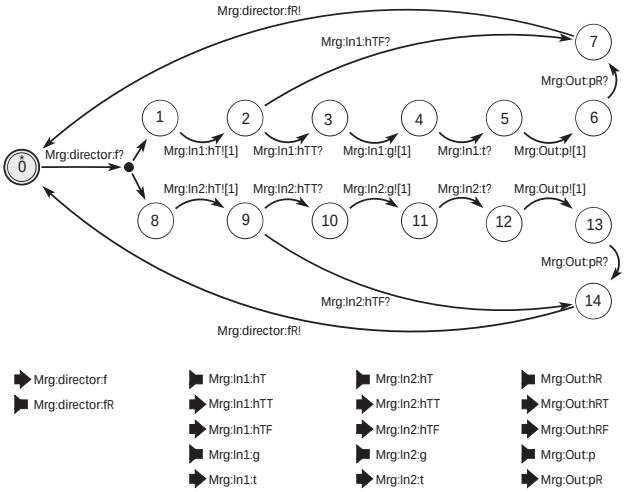


**Figure 3. The automaton of a non-deterministic merge actor.**

*In its initial state, the actor waits for a fire input action* f *from the director. Then one of the actions is chosen non-deterministically. In each action, the actor first checks whether one token is available with the* hT *action. It then waits for an answer from the director and if positive, it gets one token and puts one token to the output port. It returns to its initial state with a fire return output action* fR*.*

### 5.2 Generating Framework Automata

When generating the framework automaton of an actor model, we need to capture the behavioral type of the model of computation, the connection information and the scheduling information of the actor model into it.

First, we represent the current number of tokens in every communication channel using one persistent counter variable for each channel.

To capture the *behavioral type* of the model of computation, we separately create partial automata to handle every actor in the model. Each of these partial automata starts with an output transition to fire the actor, which leads to a state in which the actor is active and may execute, and from where an input transition leaves again which receives the fire return from the actor.

In the state where the actor is active and may execute, the composition framework must be ready to receive commands from the actor in any valid order. For this, a short transition sequence for every valid command leaves this state and returns again to it in a cycle. The valid execution order of the commands is ensured with a set of transient counters and guards on the leaving transitions.

An example of a partial automaton with a DDF model of computation is shown in Example 1 in Section 4.

The *connection information* of the original actor model is preserved using counter assignments on the automata sequence of the put command. Whenever an actor calls the put command on one of its output ports, all persistent counter variables that represent buffers in communication channels connected to this output port get incremented by the number of tokens specified by the actor.

Finally, for the *scheduling information*, we assume that an execution schedule is available in form of a finite state machine, where transitions represent actor firings. To create the framework automaton, we then take this execution schedule and replace its states with reset states of counting interface automata and its transitions with the corresponding partial automata, which we built above.

For frameworks with one of the two dataflow models of computation, which we introduced at the beginning, we make the following assumptions:

**SDF** The actor firing schedule is completely known and deterministic, which enables the composition framework to compute the size of all communication buffers and to ensure the availability of tokens prior to firing any actor. An actor is therefore not required to check the availability of tokens on its input ports or the availability of room on its output ports.

**DDF** The actor firing schedule is completely nondeterministic and there are infinite buffers in all communication channels. An actor must therefore check the availability of tokens on its input ports, but is not required to check the availability of room on its output ports.

# 6 Behavioral Type Checking

As we mentioned in Section 4.3, the question of compatibility of two Counting Interface Automata is undecidable in general. We will however show that the automata we generated in the last section exhibit special topological structures and features which make behavioral type checking decidable.

## 6.1 Topological Structures of Generated Automata

Figure 4 a) shows some examples of typical actor automata topologies. By construction, all paths between two reset states in an actor automaton are finite and each path corresponds to one possible firing of the actor.

In contrast, typical framework automata topologies have loops and therefore infinite paths between reset sates, as can be seen in Fig. 4 b). An important property of all of these loops is however, that they all start with an input transition and their execution can thus be controlled from outside. Further, any path between two reset states corresponds again to one firing of an actor.

When we build the product of all actor automata of a model and its framework automaton, the above mentioned topologies and properties lead to a product automaton, that consists of the product of the reset states of all automata, interconnected by finite paths, each corresponding to an actor firing. Some typical product automata topologies are shown in Fig. 1 c).

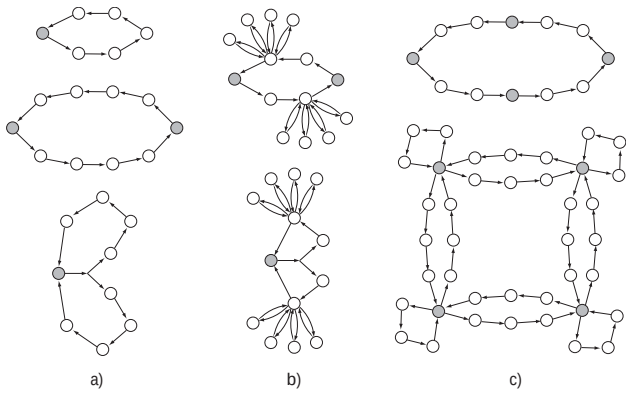


**Figure 4. Typical topological structures of a) generated actor automata, b) generated framework automata, and c) products of actor and framework automata.**

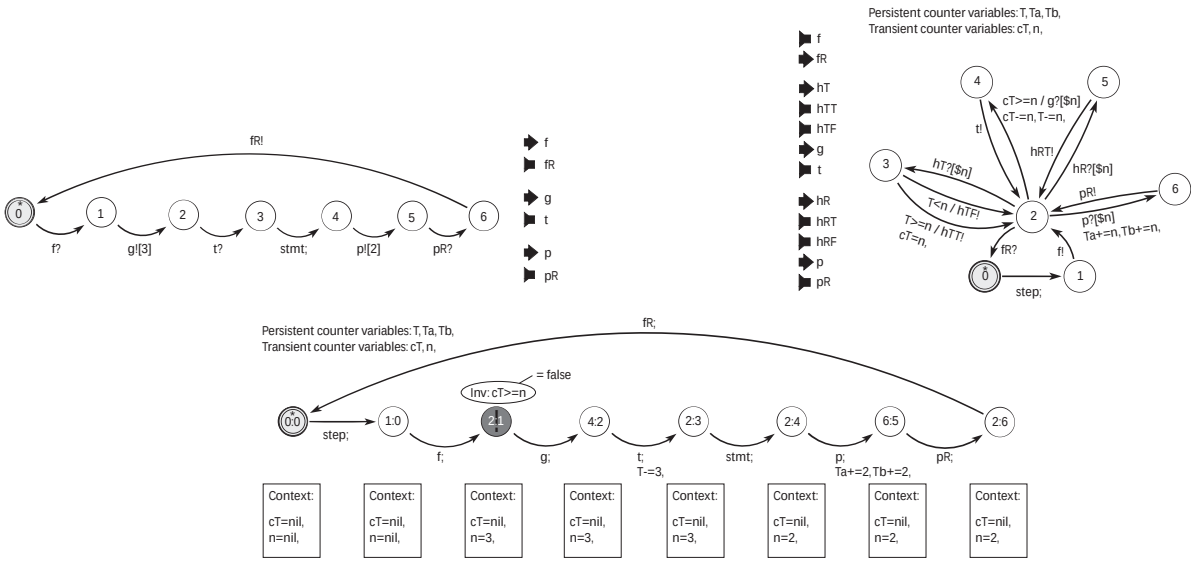


**Figure 5. The composition of a simple SDF actor with a DDF model of computation.**

## 6.2 Composition and Decidability

Let us first remind that the behavioral type of both an actor and a model of computation, expresses what behavior an actor and a model of computation expect from each other during one firing of the actor in a composition framework with a given model of computation. Thus, when we look at the generated automata from Section 5 and their typical topological structures, which were highlighted above, we see that the automata parts important for behavioral type checking, are only the state trajectories starting and ending at reset states.

Further let us remind, that when we built the framework automaton of a composition framework, we used persistent counter variables only to represent the current number of tokens in communication channels. But to express the behavioral type of the model of computation in a composition framework, we only used transient counters variables. For behavioral type checking, persistent counter variables in the automata are therefore of no importance and we can ignore their values. We must then however assume that all guards and invariants on persistent counter values may evaluate to both true and false during composition.

For behavioral type checking, we are now only caring about the values of transient counters and we know that they are reset in every reset state. Further we know, that by construction, the product of all state trajectories between two reset states are state trajectories of finite length. These properties make the composition of state trajectories between two reset states in the product automaton decidable. And since each state trajectory between two reset states in the product automaton corresponds to one actor firing in the actor model, the successful composition proofs behavioral type compatibility of the corresponding actor with the model of computation of the composition framework.

By successfully composing all state trajectories between reset states in the product automaton, we can therefore ensure behavioral type compatibility of all actors with the composition framework. We will end up with a composite actor automaton in which the internal token exchange information is contained in the previously ignored persistent counter assignments along the composed state trajectories between reset states.

**Example 4** *In this example, we check the behavioral type compatibility of a simple actor, which was implemented to run with an SDF model of computation, with a DDF model of computation. The actor consumes one token on its input port and produces one token on its output port, and since it was implemented to run with a SDF model of computation, it does not check the availability of tokens prior to consuming them.*

*Figure 5 shows the actor automaton on the top left, the relevant part of the framework automaton on the top right, and their product automaton, annotated with the context of the transient counters of each state, at the bottom. Note that the counter context of a state must already reflect action quantity declarations of leaving transitions, because guards on a transition are evaluated after the action quantity declarations on the same transition. When the actor tries to get a token without first checking its availability, the guard on the corresponding input transition of the framework automaton evaluates to false and the transition will therefore*

*not be active, leading to the error state 2:1 in the product. After pruning, the composition will be empty, showing as expected the interface incompatibility of the SDF actor with the DDF model of computation.*

## 7 Composite Actor Model Analysis

The information contained in the model automaton can be analyzed using a variety of different analysis techniques. Here, we only show as an example, how the token exchange information can be extracted from the model automaton. A case study, where Petri nets [16] are used to analyze this information is given in [18].

We can extract the token exchange information of a model automaton into another automaton, which we call the *token exchange automaton*. The states of this automaton are the reset states of the model automaton, and the transitions correspond to the paths between the reset states of the model automaton and contain all assignments to persistent counter variables along these paths. In this new automaton, the states represent the internal states of the composite actor model and each transition represents one internal actor firing, with the assignments on the transition representing the token exchange rates.

**Example 5** *Consider an actor model with feedback loops, as in Fig. 6. The actor* Src *alternatingly generates one token on its upper or its lower port, each time it is fired. The actors* FA *and* FB *are identical and both consume one token from each of their input ports and produce one token on their output port, each time they are fired. Finally, the actor* Mrg *is the non-deterministic merge actor from Ex. 2.*

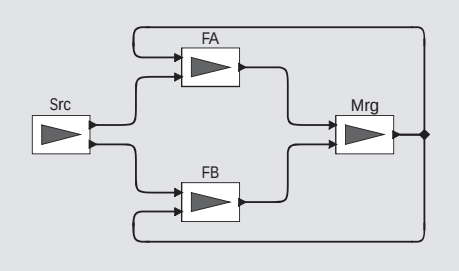


**Figure 6. An actor model with feedback.**

*After automata generation and composition of the above actor model, we will eventually get a model automaton from which the token exchange automaton shown in Fig. 7 can be extracted.*

*The analysis of this extracted token exchange information will show that the upper input port of actor FA and/or the lower input port of actor FB, will eventually overflow.*

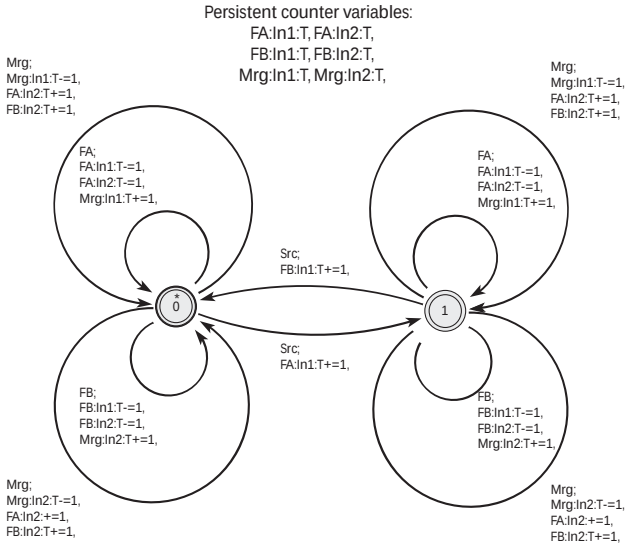


**Figure 7. The token exchange information of the actor model in Fig. 6, extracted from the composite actor automaton.**

## 8 Conclusions

We proposed a new interface theory that is capable of counting with numbers, called Counting Interface Automata (CIA). In our presented actor model analysis strategy, we used this interface theory to represent the interface information of actors and their composition framework. Thereby, its capabilities allowed us to capture temporal and quantitative aspects of an actors interface towards its composition framework and we could also capture the interconnection information, the firing schedule as well as the token exchange information of the the complete actor model.

We then presented a method to generate CIAs of actors written in CAL and for composition frameworks with an SDF or DDF model of computation.

We showed, that by composing all CIAs, we could firstly prove behavioral type compatibility of all actors with a composition framework and secondly get a single automaton that contained much information for further static analysis of the composite actor model.

## Acknowledgements

## References

[1] The Caltrop Project. http://embedded.eecs.berkeley.edu/caltrop.

[2] The Ptolemy II Project. http://ptolemy.eecs.berkeley.edu.

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

[4] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource Interfaces. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2855*, pages 117–133. Springer-Verlag, 2003.

[5] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[6] L. de Alfaro and T. A. Henzinger. Interface Theories for Component-based Design. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211*, pages 148–165. Springer-Verlag, 2001.

[7] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed Interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2491*, pages 108–122. Springer-Verlag, 2002.

[8] J. Eker and J. Janneck. Caltrop — Language report (Draft). Technical memorandum, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002. http://www.gigascale.org/caltrop.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[10] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intellingence*, 8(3):323–363, June 1977.

[11] J. W. Janneck. Actors and Their Composition. *to appear in: Formal Aspects of Computing*, 2003. preliminary version: Technical Report UCB/ERL M02/37.

[12] E. A. Lee. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, March 2001.

[13] E. A. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.

[14] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[15] E. A. Lee and Y. Xiong. A Behavioral Type System and Its Application in Ptolemy II. *to appear in: Aspects of Computing Journal, special issue on Semantic Foundations of Engineering Design Languages.*, 2003. This version: November 10, 2003.

[16] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, 1962.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction, Grenoble, France, LNCS 2304*. Springer-Verlag, April 2002.

[18] E. Wandeler. Static Analysis of Actor Networks. Technical Memorandum UCB/ERL M03/7, Swiss Federal Institute of Technology, Zurich, Switzerland, March 2003. (Diploma Thesis).