Actors and their composition

Jörn W. Janneck

Memorandum No. UCB/ERL M02/37
Electronics Research Laboratory
University of California at Berkeley
18 December 2002

Actors and their composition

Jörn W. Janneck

Department EECS
University of California at Berkeley
janneck@eecs.berkeley.edu

Abstract. Modern environments for modeling and designing concurrent computational systems increasingly support *heterogeneous* system models, which are characterized by different coordination mechanisms governing the interaction between concurrent components in different parts or at different levels of the model. These interaction semantics, also called *models of computation*, pose a major challenge to the definition of the meaning of heterogeneous models, especially if such a definition is to be independent of any specific set of models of computation, ways of describing actors, or notations for describing models.

This paper makes three main contributions. (1) It presents a framework for describing the semantics of actors and models of computation. The key notion is the concept of a model of computation as a program transformation that composes actor descriptions into a description of a composite actor. This framework is entirely independent of any specific syntax for describing actors, or any particular modeling language. (2) It uses this framework to describe properties of actor compositions and models of computation, and to classify and analyze them. (3) Finally, it discusses the implications of this theory for the design of languages for describing actors and models of computation.

1. Introduction

This work focuses on engineering languages for conceptually concurrent computational systems, and some of the issues arising from the fact that these systems are composed of subsystems with often very different kinds of interactions between their components. For example, at some level of the design, the activities in the system may be driven by asynchronous input from the environment (such as, e.g., the user pressing a key on a cell phone), while other parts of the system are expressed in terms of their data dependencies (such as signal processing algorithms) or their regular timing behavior (e.g., clocked control of sensors and actuators). The term *model of computation* is used to describe a particular set of rules that define the ways in which the components of a computational system interact.[1] A system that is best modeled using different models of computation is called *heterogeneous*.

The key challenge in constructing languages and environments that support the construction, evaluation, and implementation of these heterogeneous systems is in making sense of the composition of various kinds of components using different models of computation into a working, functional whole. Furthermore, these languages and environments should facilitate relevant analysis and efficient synthesis of the system or parts of it.

For many application areas, it is appropriate to constrain the possible kinds of components and component

[1] Cf. [EJL$^+$02], which also gives an overview over some commonly used models of computation.

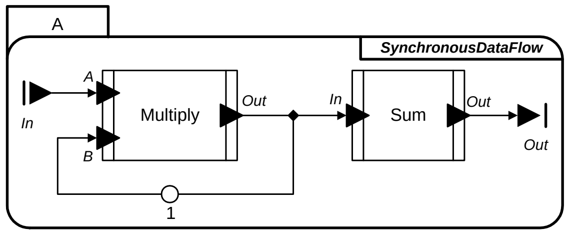

Fig. 1. A simple synchronous dataflow (SDF) model.

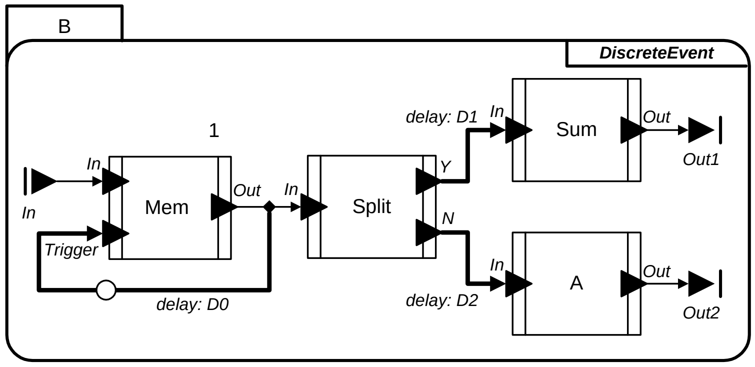

Fig. 2. A simple discrete event (DE) model. It contains the model in Fig. 1 as the actor labeled `A`.

interactions to a small number of predefined models of computation—e.g., systems such as SDL [GK97], StateCharts [Har87], or Polis [BGJ$^+$97] make the *globally asynchronous, locally synchronous (GALS)* assumption, i.e. systems modeled in these languages consist of asynchronously communicating components that function internally in a synchronous fashion. Other approaches, such as synchronous languages like Signal [ABG95] and Esterel [Ber00], allow only one kind of interaction—resulting in tools which are often ideally suited to a particular application area, but perform poorly at representing more heterogeneous systems, and say little about the interaction of different models of computation.

Fig. 1 shows such a homogeneous model. It consists of a number of simple building blocks, called *actors* (we will discuss them in a little more detail in section 2), which interact with each other according to the rules of the *synchronous dataflow* (SDF) model of computation [LM87]. For example, the components consume and produce a constant number of data items (*tokens*) each time they are activated (in the case of this model, precisely one at each input and output port), and those numbers have to be balanced such that after a predetermined sequence of component activations the buffers between components are returned to the same length.

In this work we are interested in exploring a more general approach which does not a priori constrain the kinds of component interactions, or the ways in which they are combined with each other. Examples of this kind of modeling environment would be Ptolemy [DHK$^+$01], Moses [EJ01], and Metropolis [GSV02].

We assume that models are hierarchically structured, such that parts modeled in different models of computation are nested within each other. This way of structuring heterogeneous models is called *hierarchical heterogeneity* and has been studied extensively in the Ptolemy project. [EJL$^+$02]

Consider Fig. 2, which shows a model composed under the *discrete event* (DE) model of computation. It contains the model from Fig. 1 as one of its components. The definitions of these models of computation are not relevant at this point (we will discuss them in some more detail in section 4), except that they are different—e.g., DE involves a concept of time that serves to coordinate actor execution, while in SDF it is exclusively the rates of token production and consumption that determine the execution of the actors. Another interesting aspect is the fact that the same actor, `Sum`, occurs in both models—because models of computation are not necessarily known a priori to the authors of an actor, it is desirable that actor descriptions are *polymorphic* with respect to the models of computation they can be used in.

The purpose of this work is to provide a formal framework that allows us to make sense of models such

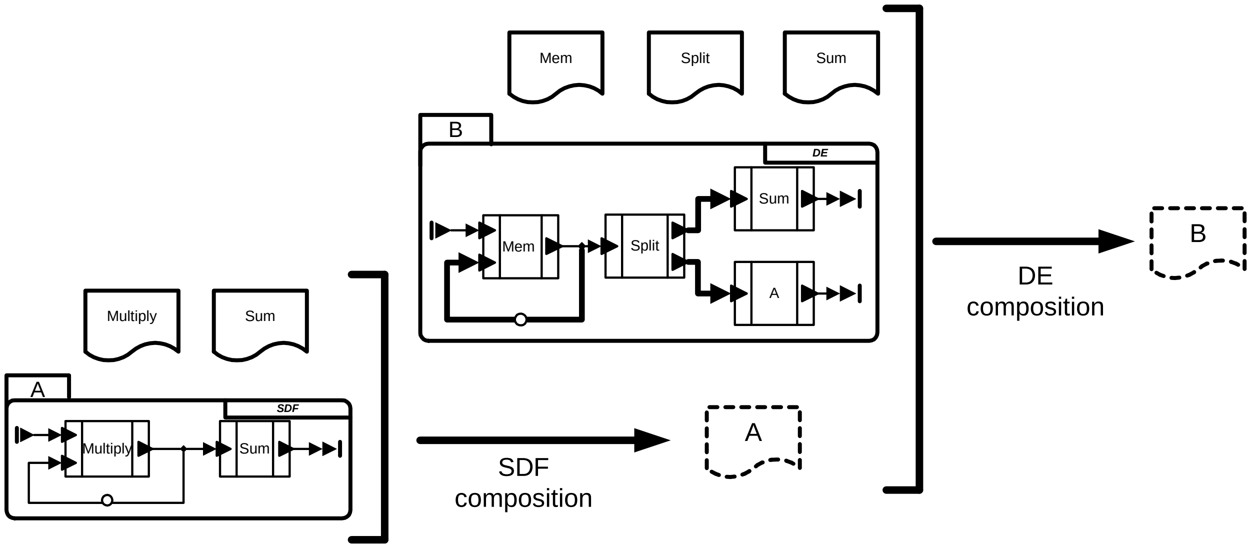

Fig. 3. Hierarchical actor composition.

as the one in Fig. 2. In particular, we want this framework to be independent of (a) any specific language for describing the basic components, (b) any specific notation for composing models from them, and (c) any specific model of computation.

The way we achieve this abstraction is by conceptualizing models of computation as actor compositions. In this view, a model of computation is a procedure which takes as input a number of actors, or more precisely their descriptions in some actor language, and some model using these actors. It then produces an actor description representing the behavior of the composite as output. This is shown in Fig. 3—the synchronous dataflow model from Fig. 1 is composed by the SDF composer, resulting in a generated actor `A`. This actor is subsequently used in the composition of the discrete event model, which is performed by a different composer, representing the DE model of computation. Since our composers work on actor descriptions in some actor language, we can consider them as (sometimes quite elaborate) program transformations.

One important consequence of viewing models of computation in some common framework such as this is that it makes it easier to study and characterize them. This is particularly relevant in a framework where models of computation are potentially created by users, and many different models of computation which have been developed independently are expected to interact meaningfully. In such a case, being able to characterize or to prove a property (absence of deadlocks, boundedness of resources or latency, etc.) for a model of computation, rather than for every or any specific model using it, may be essential to its design.

In this paper, we make the following contributions:

- We present a formal framework for actors and actor composition that does not depend on the language used to write actors or models.
- We demonstrate techniques for characterizing actors and actor composers (representing models of computation) in this framework.
- We discuss some of the implications of this framework for the design of actor languages and notations for specifying models of computation.

The following section tries to give a better intuitive understanding of our notion of actor, using a number of simple examples. Section 3 abstracts from this understanding a simple formal notion of actors. The section 4 shows how the model in Fig. 2 can be composed in the way outlined in Fig. 3, introducing some basic formal notation. In section 5 we discuss properties of models of computation, including the notion of *universal properties*, which all reasonable models of computation should exhibit. Section 6 discusses how the view of models of computation and program transformations, and the associated theory developed in the previous sections, impact the design of actor languages and, possibly, also that of languages for describing models of computation themselves. This is followed by a discussion of some related work, and some concluding remarks.

2. Describing actors

This section informally introduces the concept of an actor, and a notation for describing actors. We will need this language only for illustrating the discussion with concrete examples—none of the theory directly involves any specific language, and it is therefore sufficient to have an informal understanding of the semantics of the actor language used here.

In contrast to the visual notation we used for depicting actor compositions in Figs. 1 and 2, the actor language will be textual. While this may be a common situation, it is, however, incidental to the point of this paper, and none of the ensuing discussion rests on the specific nature of either the actor language or the notation used to express compositions.

Section 3 gives a formal definition of our notion of *actor*, but for now we may thnk of an actor as a component that communicates with its environment by sending and receiving data objects (*tokens*) via *ports*, and which performs its computation in a sequence of discrete steps, which we also call *firings* or *transitions.*

For instance, the following actor has two input ports named `A` and `B` and one output port named `Out`.

```
actor Multiply
  A, B ⟹ Out :
  action A : [a], B : [b] ⟹ Out : [a * b]
```

The `action` defines its behavior upon firing—in this case, it takes one token from each of its input ports, calls them `a` and `b`, respectively, and produces a token that is their product on its output port.

Note that the actor itself does not say whether, e.g., the incoming tokens are *consumed*, or whether the outgoing tokens are added to an input buffer of a receiving actor. This is one possibility of using the actor in the context of a model, to *interpret* it, but it will become clear later on that there are many more.

In the following, the constructs used for binding input tokens to variable names, such as `[a]` and `[b]` in the above example, are called *input patterns.* They serve as variable declarations, as well as a specification of how many input tokens are required in order to execute the respective action. Constructs defining the values of output tokens, such as `[a*b]`, are called *output expressions.*

An actor may consume and produce any number of tokens at its input and output ports, i.e. input patterns may bind more than one variable, and output expressions may compute more than one token value. For instance, the following actor reads two input tokens and produces two output tokens:

```
actor SumDiff
  In ⟹ Out :
  action In : [a, b] ⟹ Out : [a + b, a − b]
```

Both input and output tokens are ordered, from left to right. In the example, the variable a would thus be bound to the 'first' input token, b to the second, and the first output token would be their sum, the second their difference. As in the case of token consumption, it is up to the model of computation to determine what precisely this means.

The behavior of an actor can be described by more than one action. The following example shows a non-deterministic *Merge* actor:

```
actor Merge
  A, B ⟹ Out :
  action A : [x] ⟹ Out : [x]
  action B : [x] ⟹ Out : [x]
```

Its two actions copy a token from either of its two input ports to its output port. One step of this actor is the execution of one of its actions. If there are tokens available on both its input ports, both actions are *enabled*, and they may be executed in either order.[2]

Actors may also maintain internal *state*, which can be manipulated by its actions. For example, the following actor maintains (and outputs) the sum of all tokens it has read from its input port, which is sent to its output port on each firing:

[2] Strictly speaking, it depends of the model of computation whether the second action is still enabled at all after the first has been fired. It is also quite possible that the model of computation interprets this actor in a deterministic fashion. This will become clearer when we explore possible compositions in the subsequent sections.

```
actor Sum
  In ⟹ Out :
  var sum := 0
  action In : [a] ⟹ Out : [sum]
    sum := sum + a;
```

State is contained in variables, which are declared at the beginning of the actor definition. The modification of these variables is expressed by a sequence of statements, also called the *action body*, as part of the action. By convention, the values of variables used in the output expression(s) are those *after* the execution of these statements—in the example, the `sum` produced by the firing is the sum that already contains the value read from the input port.

It is important to stress that even though the language used to specify the state transition is basically an imperative language, the entire transition is nonetheless considered to be *atomic*, i.e. there is no interleaving with other action firings. As a consequence, the entire execution of the action body may be regarded as one single state transition, which is how we will model it in the next section.

When an actor has several actions, it may want to fire them depending on either its state, the value of the input tokens, or both. Our language allows to express this using *guards*, which are logical expressions attached to actions—an action may not fire unless its guards are all true. For example, the following actor copies its incoming tokens to either one of its two output ports, depending on whether a given predicate `p` is true for them:

```
actor Split
  In ⟹ Y, N :
  action In : [x] ⟹ Y : [x]
    guard p(x)
  action In : [x] ⟹ N : [x]
    guard ¬p(x)
```

Effectively, guards add additional conditions to the enabling of an action besides the presence of a certain number of tokens. These conditions may depend on the value of input tokens (as in the example above), on the state of the actor, or on both. This may cause the actor to not be able to fire in spite of a sufficient number of tokens. Depending on the model of computation, it may even deadlock the actor, because tokens that cannot be fired upon are not consumed and may block the input queues of an actor. If this is undesirable, a model of computation may declare such an actor to be inadmissible, and reject models containing it.

Actions are not required to have input patterns or produce output tokens. For example, the following actor has an action that simply stores an incoming token on one of its ports in a state variable, and reproduces that value whenever it receives a token at its other input port:

```
actor Mem
  In, Trigger ⟹ Out :
  var mem := 0
  action In : [a] ⟹
    mem := a;
  action Trigger : [x] ⟹ Out : [mem]
```

The following section will introduce a formal model for describing actors. The intention is that it will be fairly obvious how to define the semantics of our little actor language in this model, although we will not do so, because we want to focus on the general semantical framework, rather than on the meaning of a particular actor language.

3. An actor model

This section introduces a formal model for actors which will serve as the basis for the discussion of actor compositions and their properties. This model is by no means the only possible one, nor is it necessarily canonic in any sense. We have chosen this model primarily because it is easy to motivate, simple to formalize, and it is sufficient to discuss a number of interesting structures and properties of actor composition.

Corresponding properties should be easy to formulate for most reasonable extensions of this model—we hint at some of them in section 8.

Actors exchange information by sending and receiving tokens via their ports. In the following, we will assume that these tokens are elements of some universe $\mathcal{U}$ of values, which we will not further specify. As mentioned in the previous section, during one step an actor consumes and produces a number of tokens from its input ports and on its output ports. We will represent this by tuples of finite strings over the alphabet $\mathcal{U}$—e.g., the `Multiply` actor of the previous section might fire on the input tuple $([3],[4])$ producing the output tuple $([12])$, while the `SumDiff` actor would fire on the input $([7,2])$, producing the output $([9,5])$.

With this, we can define an actor with m input ports and n output ports as follows:

Definition 1 (Actor, transition). Let $\mathcal{U}$ be the *universe* of all values, and $S = \mathcal{U}^*$ be the set of all finite sequences in $\mathcal{U}$. For any non-empty set Σ of *states* an *m-to-n actor with firing* (or just *actor* for short, when m and n are understood or not relevant) is a labeled transition system

$$\langle \sigma_0, \Sigma, \tau \rangle$$

with $\sigma_0 \in \Sigma$ its *initial state*, and

$$\tau \subseteq \Sigma \times S^m \times S^n \times \Sigma$$

its *transition relation.* An element of τ is called a *transition.*

For any transition $(\sigma, s, s', \sigma') \in \tau$ we also write

$$\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'$$

or, if τ, or s and s' are understood or not relevant,

$$\sigma \xrightarrow{s \mapsto s'} \sigma' \quad \text{or} \quad \sigma \xrightarrow[\tau]{} \sigma' \quad \text{or} \quad \sigma \longrightarrow \sigma'$$

calling σ (σ') a *direct predecessor (successor)* of σ' (σ), and s (s') the *input (output)* of the transition. Together, (s, s') are the transition *label.*

The set of all m-to-n actors with firing is $\mathcal{A}^{m \longrightarrow n}$. The set of all actors is

$$\mathcal{A} =_{def} \bigcup_{m,n \in \mathbb{N}} \mathcal{A}^{m \longrightarrow n}$$

In general, actors are infinite objects—e.g., the transition relation of the `Multiply` actor would be an infinite set containing the transitions $(\cdot, (1,2), (2), \cdot)$, $(\cdot, (2,2), (4), \cdot)$, $(\cdot, (-3,15), (-45), \cdot)$, etc. (with $\cdot$ the only state of this actor). Likewise, the `Sum` actor would have a state space isomorphic to the integers, with states such as $\langle sum \leftarrow n \rangle$ for any $n \in \mathbb{Z}$, and would have transitions such as $\langle sum \leftarrow 0 \rangle \xrightarrow{2 \mapsto 2} \langle sum \leftarrow 2 \rangle$, $\langle sum \leftarrow 2 \rangle \xrightarrow{-5 \mapsto -3} \langle sum \leftarrow -3 \rangle$, $\langle sum \leftarrow 7 \rangle \xrightarrow{11 \mapsto 18} \langle sum \leftarrow 18 \rangle$ etc.

This is why we need an actor language to allow finite descriptions of actors such as the one introduced in the previous section. The following notation will allow us to discuss actor descriptions and their semantics, i.e. the actor they describe:

Definition 2 (Actor language, semantics). We will call a set $\mathcal{L}$ of well-formed *programs* denoting actors an *actor language*, and a function

$$[\![\cdot]\!] : \mathcal{L} \longrightarrow \mathcal{A}$$

its *semantics.*

Our actors are generalizations of the ones described in [Lee97] in several ways. First of all, they have state. Second, they allow non-determinism, i.e. given a state σ and an input tuple s, there may be multiple output tuples s' and s'' and/or multiple successor states σ' and σ'' such that $\sigma \xrightarrow{s \mapsto s'} \sigma'$ and $\sigma \xrightarrow{s \mapsto s''} \sigma''$. Finally, they are more liberal with respect to the permissible input tuples.[3]

In many cases, a model of computation requires actors to have specific properties—e.g., synchronous dataflow assumes that actors have constant token production and consumption rates. In this case, a model

[3] Technically, [Lee97] requires input tuples not to be *joinable*, whereas our definition has no such constraint.

containing actors that do not have these properties would not be *well-formed*, and an attempt to compose such a model would fail.

4. Composition as program transformation

In this section we elaborate the concept of actor composition as program transformation by first looking at the small examples from Figs. 1 and 2, and then introducing some formal notation for talking about models, modeling languages, and composers.

4.1. A heterogeneous example

In order to perform the composition represented in Fig. 3, we first need to compose the synchronous dataflow model from Fig. 1 into an actor `A`. The SDF composer will turn one complete cycle of the model into the atomic firing of the composite actor.

Obviously, one complete cycle of this model consists of one firing of the `Multiply` actor, followed by one firing of the `Sum` actor. The result produced by the `Multiply` actor is fed to the `Sum`, and is also stored in the buffer of the feedback loop around the `Multiply` actor, from where it will be used in the next cycle.

The description of the composite actor generated from the two component actors might look like this:

```
actor $A$
  $In \Longrightarrow Out$ :
  var $buf = 1, Sum\$sum = 0$
  action $In : [a] \Longrightarrow Out : [Sum\$sum]$
    $buf := a * buf$;
    $Sum\$sum := Sum\$sum + buf$;
```

Its action is compiled from the actions of the components, it contains their state variables (appropriately renamed to avoid name conflicts—thus the *sum* variable in the `Sum` actor becomes $Sum\$sum$), and possibly additional variables (here one representing the buffer, initialized to a value specified in the model in Fig. 1).

The discrete event composition of the model in Fig. 2 is quite a bit more involved than the synchronous dataflow composition. In part this is because SDF models can be statically scheduled, hence the generated code does not have to check any conditions at runtime. Also, our discrete-event model of computation has a local notion of time—some of its connections are labeled with *delays*. The interpretation is that tokens sent along those connections will be delayed by the appropriate amount of time before they become visible to the receiver. Effectively, they are turned into *events*, which are sorted according to their time stamps.[4]

```
actor $B$
  $In \Longrightarrow Out1, Out2$ :
  var $Mem\$s = 0, Sum\$s = 0, A\$buf = k, A\$Sum\$s = 0,$
    $q1 = [], q2 = [],$
    $events = [(time = 0, proc = event0, val = nil)], currentTime = 0$
  action $In : [a] \Longrightarrow$
    $Mem\$s := a$;
  action $\Longrightarrow$
    guard $events \neq []$
    var $e = hd(events)$
    $currentTime := e.time$;
    $events := tail(events)$;
    $e.proc(e.val)$;
  action $\Longrightarrow Out1 : [hd(q1)]$
    guard $q1 \neq []$
```

[4] A more global notion of time, which would be shared across composer boundaries, would require a timed actor model. While this is a fairly straightforward extension to our model in section 3, we have chosen an untimed model for simplicity.

```
16     $q1 := tail(q1);$
17   **action** $\Longrightarrow Out2 : [hd(q2)]$
18     **guard** $q2 \neq []$
19     $q2 := tail(q2);$
20   **procedure** $event0(x)$
21     $queue(event0, D0, x);$
22     **if** $p(x)$
23       $queue(event1, D1, x);$
24     **elseif** $\neg p(x)$
25       $queue(event2, D2, x);$
26   **procedure** $event1(a)$
27     $Sum\$s := Sum\$s + a;$
28     $q1 := q1 + [Sum\$s];$
29   **procedure** $event2(a)$
30     $A\$buf := a * A\$buf;$
31     $A\$Sum\$a := A\$Sum\$s + A\$buf;$
32     $q1 := q1 + [A\$Sum\$s];$
33   **procedure** $queue(p, d, v)$
34     $events := insert((time = currentTime + d, proc = p, val = v),$
35     $events);$
```

Just as in the case of the SDF composition, the DE composite has a number of state variables that correspond to state variables of the components, which are declared in line 3. Line 4 contains variables whose values represent the output queues—these are filled by executing code which corresponds to the internal actors connected to output ports, for example the procedure corresponding to the code of actor `A` in lines 29 to 32, and they are emptied by output actions, defined in lines 14 to 19. Input is handled by an input action, in lines 6 to 7.[5] An interesting aspect of this composition is that it also introduces an action with neither input patterns nor output expressions—lines 8 to 13. This action executes internal actor firings whenever there are pending events in the event queue.

The important message that this example is intended to illustrate is that the actors generated by different composers can be quite different, both in the way they organize code and in the way their state is assembled. Some composers may simply paste together actions, using their input patterns and output expressions to join them via, e.g., commonly used variables. Others may introduce significant administrative 'glue', involving coordinating code as well as additional state. Some models of computation may even maintain several copies of the state of the component actors, in order to roll back computation to an earlier state. In the following section we show how to discuss composers and the structures they create from component actors somewhat more formally and abstractly. In order to do this, we first have to provide some notation for talking about models and composers.

4.2. Models and composers

The main motivation for the following definitions is to abstract from specific notations for describing actors and models—while it is essential that there is a language for describing actors, and possibly several notations for building models, our goal here is to introduce abstractions that are applicable to a broad range of possible syntaxes. The previous section does this for actors. Here, we do this for models, trying to capture the following salient points of any actor-based modeling language:

1. A model is an arbitrary syntactical structure.
2. Different models of computation may apply to different model structures/syntaxes.
3. A model contains references to actors.
4. It may also contain other information, such as how actors interact, which role individual actors play in the composition, etc.

[5] Note that in this composition we make use of the simplifications possible when all component actions only read a single token—otherwise, we would need additional buffers, which would only complicate the presentation here.

5. There is a notion of *well-formedness* of models. A well-formed model can be meaningfully composed under a given model of computation.

We distinguish in the following definition between a model structure and a model. The former will contain uninterpreted references to actor names—e.g., Figs. 1 and 2 would depict model structures. By contrast, a model is a model structure augmented with interpretations of the actor names—e.g. Fig. 1 together with the definitions of the actors `Multiply` and `Sum`.

Definition 3 (Model, model structure). Given a modeling language $\mathcal{M}$, and a set of symbols $\mathcal{V}$, each *model structure* $M \in \mathcal{M}$ is associated with a finite set of *actor variables* $V_M \subset \mathcal{V}$. A *model* M_α is a pair

$$\langle M, \alpha \rangle$$

with $\alpha : V_M \longrightarrow \mathcal{L}$ a valuation of each actor variable in M with an actor description in the language $\mathcal{L}$.
The set of all models in $\mathcal{M}$ is called $\bar{\mathcal{M}}$.

We say that a model of computation is represented or implemented by a composer. A composer is simply a partial map from a set of models $\bar{\mathcal{M}}$ to an actor language, $\mathcal{L}$.

Definition 4 (Composer/composition function, well-formed models). A *composer* or *composition function* $\mathcal{C}$ in $\bar{\mathcal{M}}$ is a partial function

$$\mathcal{C} : \bar{\mathcal{M}} \rightharpoonup \mathcal{L}$$

A model M_α is *well-formed* under $\mathcal{C}$ iff $M_\alpha \in \mathbf{cor}\, \mathcal{C}$, i.e. if $\exists A \in \mathcal{L} : \mathcal{C}(M_\alpha) = A$.

Well-formedness of a model, involving diagnosing, locating, and reporting errors, as well as describing the well-formedness rules for models in the first place, is of course a very important issue in practice. However, since this work focuses on the semantical aspects of actor composition, well-formedness of models is beyond the scope of this paper.[6]

5. Properties of models of computation

In this section we examine the structure of composers, formulate some fundamental properties, and characterize different kinds of composers. This is by no means an exhaustive list of properties—it is primarily intended to showcase some of the more typical uses of the formal structures we have introduced previously, and to illustrate some of the ways in which we may now talk about, analyze and characterize models of computation.

5.1. Finite iterations and actor morphisms

In the following, we are interested in the relation between a firing of the composite actors and the firings performed by its components. In general, a component can make any number of steps during a single step of the composite, as long as it is a finite number of steps. In order to make it easier to talk about this, we will construct from a component actor A another actor A^* which has the property that every finite sequence of steps in A is one single step in A^*. We call this actor the *iteration* of A.

Definition 5 (Iteration). Given an actor $A = \langle \sigma_0, \Sigma, \tau \rangle$, we define a *(finite) iteration* $\tau^n \subseteq \Sigma \times S^m \times S^n \times \Sigma$ of its transition relation as follows, using the same notational convention, i.e. we write $\sigma \xrightarrow[\tau^n]{s \mapsto s'} \sigma'$ for $(\sigma, s, \sigma', s') \in \tau^n$. For any $n \in \mathbb{N}$, τ^n is the smallest set such that:

$$\sigma \xrightarrow[\tau^0]{\lambda \mapsto \lambda} \sigma \qquad \text{for all} \quad \sigma \in \Sigma$$

$$\exists \sigma'' : \sigma \xrightarrow[\tau^{n-1}]{u \mapsto u'} \sigma'' \wedge \sigma'' \xrightarrow[\tau]{s \mapsto s'} \sigma' \Longrightarrow \sigma \xrightarrow[\tau^n]{us \mapsto u's'} \sigma' \qquad \text{for} \quad n > 0$$

[6] See, e.g., [JE01] for some discussion of these issues for visual modeling languages.

The set $\tau^* =_{def} \bigcup_{n\in\mathbb{N}} \tau^n$ is the union of all finite iterations of τ.

We call the actor

$$A^* =_{def} \langle \sigma_0, \Sigma, \tau^* \rangle$$

the *finite iteration of A*. The set of all finite iterations is called $\mathcal{A}^*$:

$$\mathcal{A}^* =_{def} \{A^* \mid A \in \mathcal{A}\}$$

Working with A^* instead of A means that we allow A to perform any finite number of steps, including none at all. Note that the actors in $\mathcal{A}^*$ are their own interations, i.e. for each $A \in \mathcal{A}$, $A^* = (A^*)^*$.

When investigating the relation between, e.g., the composite actor and the iterations of its components, we establish a mapping from one to the other—more precisely, we establish two mappings: one between their state spaces, and another one between their transition relations. We call such a pair of mappings a *morphism*, and define it as follows:[7]

Definition 6 (Actor morphism). Given two actors $A = \langle \sigma_A, \Sigma_A, \tau_A \rangle$ and $B = \langle \sigma_B, \Sigma_B, \tau_B \rangle$, an *actor morphism*

$$\phi : A \longrightarrow B$$

is a pair of functions $\langle s, t \rangle$, with $s : \Sigma_A \longrightarrow \Sigma_B$ and $t : \tau_A \longrightarrow \tau_B$ such that the following hold:

1. $s(\sigma_A) = \sigma_B$
2. $t(\sigma, v, w, \sigma') = (s(\sigma), v', w', s(\sigma'))$ for some v', w'.

For clarity, we will sometimes use the name of the morphism as a subscript for s and t.

Intuitively, the t mapping between the transition relations respects the s mapping between the state spaces. Morphisms in general can do arbitrary things to the labels of transitions, although it does make sense to identify classes of morphisms that are more constrained than that.

A few simple examples for actor morphisms are the following:

1. For any actor A, its *identity morphism* is defined by the two identity functions on state space and transition relation, and is called 1_A.
2. For any actor A, there exists a unique morphism $\iota_A : A \longrightarrow A^*$, called its *injection into its iteration*, which is defined as follows:

$$\begin{aligned} \iota_A &= \langle s, t \rangle \\ s &: \sigma \mapsto \sigma \\ t &: (\sigma, v, w, \sigma') \mapsto (\sigma, v, w, \sigma') \end{aligned}$$

3. Assume we have an actor $T = \langle *, \{*\}, \{(*, \lambda, \lambda, *)\} \rangle$. Then for any actor A, there is precisely one morphism $T_A : A \longrightarrow T$, and it is defined as

$$\begin{aligned} T_A &= \langle s, t \rangle \\ s &: \sigma \mapsto * \\ t &: (\sigma, v, w, \sigma') \mapsto (*, \lambda, \lambda, *) \end{aligned}$$

5.2. Decomposition

In order to analyze the relation between a composite and its components, it is necessary to disassemble the composite into its parts. For this reason we assume that each actor composer can be associated with a decomposition, defined as follows.

Definition 7 (Actor decomposition). Given an actor A and a finite set of *component actors* $\{A_v \mid v \in$

[7] Clearly, actors and actor morphisms form a category. However, since we will only need the most basic categorial concepts, little would be gained from using categorial terminology here.

$$\begin{tikzcd}
\mathcal{L} \arrow[d, "{[\![\cdot]\!]}"'] & \bar{\mathcal{M}} \arrow[l, harpoon', "\alpha_v"'] \arrow[r, harpoon, "\mathcal{C}"] & \mathcal{L} \arrow[d, "{[\![\cdot]\!]}"] & \\
\mathcal{A} \arrow[r, hook, "\iota"] & \mathcal{A}^* & \mathcal{A} \arrow[l, "\phi_v"'] \arrow[r, "\phi^R"] & \mathcal{A}
\end{tikzcd}$$

Fig. 4. The sets of objects and their key relations and functions involved in actor composition.

$V\}$, a *decomposition* of A into the A_v with *residue* A^R is a set of actor morphisms $\phi_v : A \longrightarrow A_v^*$ and $\phi^R : A \longrightarrow A^R$ such that t^R is defined as

$$t^R : (\sigma, w, w', \sigma') \mapsto (s^R(\sigma), w, w', s^R(\sigma'))$$

We write a decomposition as $\langle (\phi_v)_{v \in V}, \phi^R \rangle$.

To illustrate decomposition by an example, take the composite actor `A` from section 4, representing the model in Fig. 1. It has states such as $\langle buf \leftarrow 1, Sum\$sum \leftarrow 0 \rangle$ (its initial state), $\langle buf \leftarrow 12, Sum\$sum \leftarrow 15 \rangle$, etc. It has three decomposition morphisms, $\phi_{Multiply}$, ϕ_{Sum}, and the residual ϕ^R. Let us look at them in turn.

The state mapping $s_{Multiply}$ returns the same state $\cdot$ for any composite state $\langle buf \leftarrow x, Sum\$sum \leftarrow y \rangle$, because the `Multiply` actor is stateless (i.e. it has only one state). The transition map $t_{Multiply}$ is defined as follows:

$$\langle buf \leftarrow x, Sum\$sum \leftarrow y \rangle \xrightarrow{[n] \mapsto [y+xn]} \langle buf \leftarrow xn, Sum\$sum \leftarrow y + xn \rangle$$
$$\mapsto \cdot \xrightarrow{[n],[x] \mapsto [xn]} \cdot$$

It is easy to see that this only produces transitions that are valid in `Multiply`.

The state mapping s_{Sum} is defined as

$$\langle buf \leftarrow x, Sum\$sum \leftarrow y \rangle \mapsto \langle sum \leftarrow y \rangle$$

while the transition mapping t_{Sum} is defined as

$$\langle buf \leftarrow x, Sum\$sum \leftarrow y \rangle \xrightarrow{[n] \mapsto [y+xn]} \langle buf \leftarrow xn, Sum\$sum \leftarrow y + xn \rangle$$
$$\mapsto \langle sum \leftarrow y \rangle \xrightarrow{[xn] \mapsto [y+xn]} \langle sum \leftarrow y + xn \rangle$$

Finally, for the residual decomposition morphism, the state mapping s^R (which also determines the transition mapping) is defined as

$$\langle buf \leftarrow x, Sum\$sum \leftarrow y \rangle \mapsto \langle buf \leftarrow x \rangle$$

The existence of a decomposition for a given composite actor created by a composer is not a trivial property, as we will see below. If we want to make use of this in practice, it means that the composer has to be described in such a way so that properties of its decomposition, or possibly some description of the decomposition itself, can be inferred from it.

Note that a decomposition describes relations between actors, in contrast to the composer, which is an operation on actor *descriptions*. This means that decompositions are in general infinite objects—as a consequence, we will either have to introduce a notation for finitely representing decompositions, or we have to abstract them into a set of properties which are relevant to a particular task or proof, and which we can infer from the composer and the composition. We address this issue shortly when we discuss composer languages in section 6—for now, we will assume that we know the decomposition for every composite of a model of computation.

Fig. 4 gives an overview of the various sets of objects that we are talking about here, and how they are related. It is best read starting from the center of the top row, where we have the set of models (assuming a particular modeling language), $\bar{\mathcal{M}}$. α_v is a family of partial functions indexed by the variable symbols: if a model M_α contains variable v, then $\alpha_v(M_\alpha) = \alpha(v)$, otherwise it is undefined. The $\mathcal{C}$ is a composer that maps the well-formed models in $\bar{\mathcal{M}}$ to a description of their composite in $\mathcal{L}$.

Both the composite description as well as the description of the components are mapped by the interpretation $[\![\cdot]\!]$ to the corresponding actors. The components are injected into their iteration, while the composite is taken apart by the appropriate decomposition.

5.3. Safety

The first property of a composition that we will be looking at concerns the preservation of unreachable states. The idea here is that a component actor should not be able to reach *more* states due to being embedded into a context. A composition that ensures this is called *safe*. First we need to define the reachable states of an actor.

Definition 8 (Reachable states/transitions). Given an actor $\langle \sigma_0, \Sigma, \tau \rangle$, and some state $\sigma \in \Sigma$, the set $\bar{\Sigma}_\sigma \subseteq \Sigma$ of *σ-reachable states* the set of all σ' such that there exists a finite sequence $(\sigma_i)_{i=1..k}$, $k \geq 1$, such that

1. $\sigma = \sigma_1$
2. $\sigma_i \xrightarrow[\tau]{} \sigma_{i+1}$ for $i = 1..k-1$
3. $\sigma_k = \sigma'$

A state is *reachable* if it is σ_0-reachable. The set of all reachable states is $\bar{\Sigma}$. We call a transition $(\sigma', v, w, \sigma'') \in \tau$ *σ-reachable* iff σ' is σ-reachable. It is *reachable* if it is σ_0-reachable.

Safety is a very basic property, which in fact follows from the *existence* of the decomposition alone.

Theorem 1 (Safety). A composite actor $A = \langle \sigma_0, \Sigma, \tau \rangle$ is *safe* if its decomposition $\left\langle (\phi_v)_{v \in V}, \phi^R \right\rangle$ has the property that for all reachable states σ of A and all $v \in V$, $s_{\phi_v}(\sigma)$ is reachable in A_v.

A composer is safe if all the composite actors in its range are.

First of all, note that in any finite iteration A^*, $\sigma_0 \xrightarrow[\tau^*]{} \sigma$ means that there is some k such that $\sigma_0 \xrightarrow[\tau^k]{} \sigma$. This means that if σ can be reached in one step in A^*, it is reachable in A. The converse is also true: If σ is reachable in A, there is a k such that it can be reached from σ_0 in k steps, and hence $\sigma_0 \xrightarrow[\tau^k]{} \sigma$, and thus it is reachable in A^* in one step. Furthermore, it is easy to see that any reachable state in A^* is reachable in a single step. Hence we have the following lemma:

Lemma 1. For any actor A, σ is reachable in A if and only if it is reachable in A^*.

We can now prove the theorem as follows:

Proof. Since σ is reachable in A, there is a sequence of states $(\sigma_i)_{i=1..k}$ that leads from σ_0 to σ. ϕ_v is a morphism—this means that the sequence $(s_{\phi_v}(\sigma_i))_{i=1..k}$ is a sequence of states connected by transitions in A_v^* that leads from $\sigma_{0,v} = s_{\phi_v}(\sigma_0)$ to $s_{\phi_v}(\sigma)$. In other words, $s_{\phi_v}(\sigma)$ is reachable in A_v^*. Therefore, by the lemma, it is reachable in A_v. $\Box$

This property follows from the fact that actor morphisms respect the transition structure of the actors they relate, and the fact that the decomposition is a collection of such morphisms. As a consequence, safety of a composer can be guaranteed by providing a suitable decomposition for each composite actor in its range.

Safety is such a fundamental property of a composition, that it seems reasonable to make it a *requirement* for a composition to be safe. We call such a property *universal*, without giving a formal definition of this term—in fact, it seems likely that there is no formal criterion for whether a property is universal, but rather that the decision about the fundamental requirements on composers is simply part of designing a framework for describing models of computation.

5.4. Boundedness

Boundedness is an example of a property of a composite actor that depends on the structure of its transition relation. Each step of the composite actor involves a number of steps of each component. It is bounded if there is an upper bound as to how many steps any of its components takes for any step of the composite. First, we need to define the *length* of a transition in the iteration of an actor, since the decomposition maps into the iteration of the components, rather than the actor itself.

Definition 9 (Transition length). Given an actor $A = \langle \sigma_0, \Sigma, \tau \rangle$. For each transition $y \in \tau^*$ of its iteration A^*, its *length* $\mid y \mid_A$ in A is the smallest number k such that $y \in \tau^k$. (Cf. Def. 5.)

With this we can easily define boundedness as follows.

Definition 10 (Boundedness). A composite actor $A = \langle \sigma_0, \Sigma, \tau \rangle$ is *bounded* if its decomposition $\left\langle (\phi_v)_{v \in V}, \phi^R \right\rangle$ has the following property:

$$\exists k \in \mathbb{N} : \forall v \in V, y \in \tau : y \text{ reachable} \Rightarrow \mid t_{\phi_v}(y) \mid_{A_v} \leq k$$

A composer is *bounded* if all the composite actors in its range are.

Boundedness is an important ingredient of many interesting performance characteristics of a system. For instance, boundedness in the above sense is a necessary condition for the existence of an upper bound for 'latency', i.e. the time it takes an actor to make one step.[8]

5.5. Liveness

A component in a composition is *dead* in some state σ if there is no subsequent transition so that it 'does something' in that transition. We capture this idea in the following definition.

Definition 11 (Liveness). A component actor A_v in a composite $\langle \sigma_0, \Sigma, \tau \rangle$ is *σ-live* in some state $\sigma \in \Sigma$ iff for all reachable σ' there exists a σ'-reachable $y \in \tau$ such that $t_v(y) \notin \tau_v^0$. It is *live* if it is σ_0-live and it is *(σ-) dead* if it is not (σ-) live.

However, actor composition is a very flexible notion, and there may be useful compositions where some or all actors appear to be dead. As an example, consider a composition that allows for rollback of the component actors (as might be the case, e.g., for a model of computation that realizes a protocol similar to Time Warp [Jef85]). Say, in a step $\sigma \xrightarrow[\tau]{} \sigma'$ of the composite, a component actor A_v would make a step $\sigma_v \xrightarrow[\tau_v]{} \sigma'_v$. If the step of the component might be rolled back, the decomposition morphism $\phi_v = \langle s_v, t_v \rangle$ must have the property that $s_v(\sigma) = s_v(\sigma')$, because in general there need not be an inverse transition $s_v(\sigma') \xrightarrow[\tau_v]{} s_v(\sigma)$ in the (iteration of the) component.[9] In general, then, for any state σ, $s_v(\sigma)$ is the earliest state that the component A_v may be rolled back to. In practice, a composite actor might keep copies of its components' states as part of the residual A^R, but it may also just start computation from $s_v(\sigma_0)$ each time it makes a step $\sigma \xrightarrow[\tau]{} \sigma'$.

In the most extreme case, the actors A_v of the decomposition may never advance, i.e. $s_v(\sigma)$ maps to the initial state of A_v for any σ, and $t_v(y) \in \tau_v^0$ for any transition $y \in \tau$. In other words, as far as the decomposition is concerned, the component actors never make any progress. This is, of course, not a deficiency of the composition mechanism. Instead it shows that using a simple decomposition as defined in Def. 7 is too coarse a tool for capturing the subtleties of some models of computation.

5.6. Dataflow models of computation

We will now use our framework to characterize dataflow models, one important and interesting class of models of computation with many more or less specialized instances. [LP95] The main purpose is to illustrate the use of our framework for describing structures of models of computation. These structures may be used to

[8] This assumes, of course, that there is a minimal amount of work to be done in each actor firing, so that an unbounded number of firings of the component actors implies an unbounded latency for the composite actor.

[9] This is of course a consequence of the fact that ϕ_v is a morphism as defined in Def. 6: Assume, e.g., that some $\sigma' \xrightarrow[\tau]{} \sigma''$ rolls back the component, i.e. $s_v(\sigma'') = s_v(\sigma)$. Then by the second property of morphisms, there needs to be a transition $s_v(\sigma') \xrightarrow[\tau_v]{} s_v(\sigma)$.

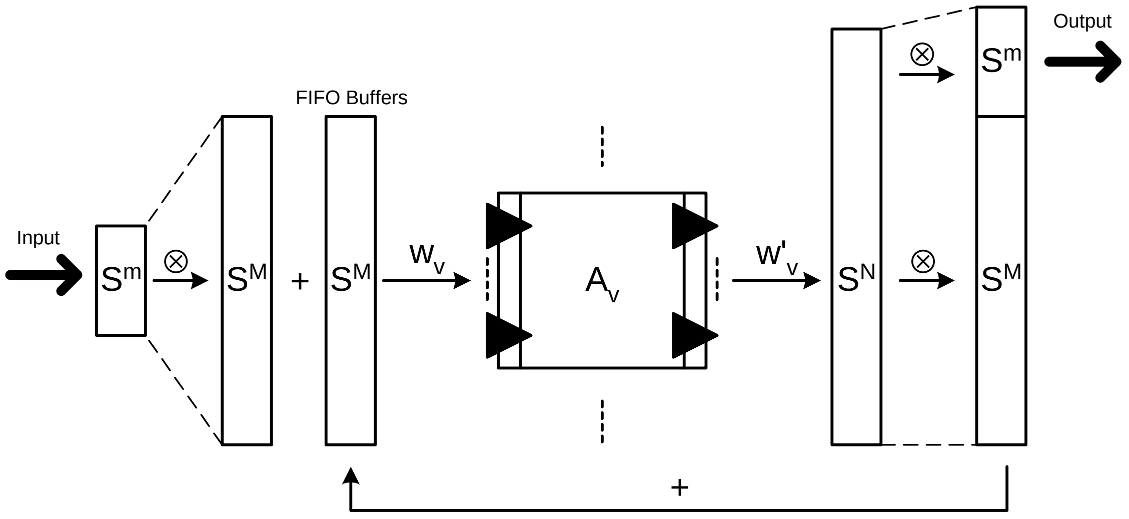

Fig. 5. Flow of data in transitions from state σ in a dataflow model of computation.

classify models of computation, but perhaps an even more interesting application is to use them as a starting point for designing composer languages—we will discuss this a little more in the next section.

Dataflow models are typically characterized by component actors communicating with each other via FIFO queues associated with each actor input port—tokens produced at the output ports of actors are added to the end of these queues, while input tokens are read from the head of the queues and are consumed, i.e. they will not be available for another firing.

In order to characterize dataflow models of computation in the framework that we developed here, we need to (a) identify the buffers as part of the residual state, and (b) describe how the component actors read tokens from and write tokens into those buffers.

Let the composite actor be $A \in \mathcal{A}^{m \longrightarrow n}$ and its component actors $A_v \in \mathcal{A}^{m_v \longrightarrow n_v}$. We define $M = \sum m_v$ and $N = \sum n_v$. Because we associate a FIFO buffer with each input port, the state of those buffers is an element in S^M.

Fig. 5 illustrates the basic flow of information during a firing of a dataflow model. The input token sequences, which together are an element in S^m, are distributed and added to the (end of the) current buffers. The actors consume prefixes of 'their' buffers, and collectively produce an output tuple of sequences in S^N. This collective output is distributed and added to the buffers, and some of it is also sent to the output ports of the composite actor. Note that in most cases, these activities are interleaved—for example, some component actors fire, and their output is added to the buffers, and only then other component actors can fire and so on. The figure merely shows the net effect, abstracting from the *scheduling* of actor firings and token distribution, which are of course of paramount importance in any specific dataflow model of computation. For characterizing dataflow models of computation collectively, however, this abstraction is more appropriate.

As a prerequisite for describing Fig. 5 formally, we need some notation for composing the input tuples and output tuples of sequences consumed and produced by the component actors into the larger tuples in S^M and S^N, respectively. We assume there is some unique combinator $\bigcirc x_v$ that takes tuples x_v (we always assume $v \in V$) and produces a tuple the size of the sum of the sizes of the x_v.

Another operation that is used in the figure is the distribution of a tuple of sequences over another tuple of sequences. Each sequence in the result tuple must be an *order-preserving merge* of sequences in the original tuple. Intuitively, this means that the order of tokens inside the sequences may not be altered, and sequences must be used either completely, or not at all. E.g., for a 3-tuple of sequences $(abc, def, ghij)$ the sequences $abdefc$, λ, $defabc$, abc, and $ghabijcdef$ are all order-preserving merges, but ab, cba, $aabc$, bc are not. We write $\bigotimes w$ for the set of all order preserving merges of a tuple of sequences w and $\bigotimes^k w$ for the set of all k-tuples of sequences such that each sequence is in $\bigotimes w$.

We can now describe dataflow models of computation as follows.

Definition 12 (Dataflow compositions/composers). A composite actor $A \in \mathcal{A}^{m \longrightarrow n}$ with decomposition $\langle (\phi_v)_{v \in V}, \phi^R \rangle$, with $A_v \in \mathcal{A}^{m_v \longrightarrow n_v}$ and $M = \sum m_v$ and $N = \sum n_v$, is a *dataflow composition* if there is a function $\beta : \Sigma^R \longrightarrow S^M$, such that for each transition $\sigma \xrightarrow{w \mapsto w'} \sigma'$ of A (and $t_v(\sigma, w, w', \sigma') = (s_v(\sigma), w_v, w'_v, s_v(\sigma')))$ there exist $i, b \in S^M$ such that the following holds:[10]

1. $i \in \bigotimes^M w$
2. $b \in \bigotimes^M \bigcirc w'_v$
3. $w' \in \bigotimes^n \bigcirc w'_v$
4. $\beta \circ s^R(\sigma) + i + b = \bigcirc w_v + \beta \circ s^R(\sigma')$

A composer is a *dataflow composer* if each actor in its range is a dataflow composition.

The first condition states that i is an M-tuple of order-preserving merges of the input sequences, the second and third state that b and w' are tuples of order-preserving merges of the output sequences of the component actors. The fourth condition is the interesting one—it requires that the original buffer state concatenated with the input tokens in i and the output of the component actors b is identical to the input of the component actors concatenated with the new buffer state. This is the *dataflow condition*, which describes how tokens are consumed from and added to the FIFO queues of the composite actor.

As an example, we may again look at the composite actor `A` from section 4—since it has been created by an SDF composition, it is certainly a dataflow composite. Based on the discussion of its decomposition in section 5.2, we will now look at how it fits our definition of dataflow models of computation.

First of all, the two component actors have three input ports, A and B (of the `Multiply` actor), and In (of the `Sum` actor). Hence we have three FIFO buffers, which we will write in the above order. The next step is to define the mapping from the residual state to a buffer comfiguration. Above we defined the residual state to be mapped from a composite state as follows:

$s^R : \langle \mathit{buf} \leftarrow x, \mathit{Sum\$sum} \leftarrow y \rangle \mapsto \langle \mathit{buf} \leftarrow x \rangle$

Based on this, we define β like this:

$\beta : \langle \mathit{buf} \leftarrow x \rangle \mapsto ([], [x], [])$

In other words, in each state, the buffers for the A and In ports are empty, and the one for the B port contains exactly one token, the value of the buf variable. The reason why we do not need to represent the other buffers as part of the state, and why we can assume that the B buffer contains only one value is, of course, due to the constraints of the SDF model of computation.

Now we need to find the i and b for an arbitrary transition of the composite. In this simple case, every transition is of the form

$$\langle \mathit{buf} \leftarrow x, \mathit{Sum\$sum} \leftarrow y \rangle \xrightarrow{n \mapsto xn+y} \langle \mathit{buf} \leftarrow xn, \mathit{Sum\$sum} \leftarrow xn + y \rangle$$

.

Now we choose $i = ([n], [], [])$ and $b = ([], [xn], [xn])$. The first condition is certainly true, $([n], [], []) \in \bigotimes^3([n])$. Since $\bigcirc w'_v = ([xn], [xn + y])$ the second and third condition (with $w' = ([xn + y]))$ are also easily seen to be fulfilled. The fourth condition boils down to the equation

$$([], [x], []) + ([n], [], []) + ([], [xn], [xn]) = ([n], [x], [xn]) + ([], [xn], [])$$

Note that the last tuple represents the buffer configuration in the subsequent state, as per our definition of β. As both sides of the equation evaluate to $([n], [x, xn], [xn])$, the dataflow condition is fulfilled as well, and hence `A` is a dataflow composition.

The above definition of dataflow is very broad (e.g., it does not even require causality—tokens may be consumed before they are produced, as long as the overall dataflow condition is fulfilled at the 'end' of the transition of the composite actor), and for many practical purposes we might want to have more specific knowledge about a dataflow model. Extending this analysis any further, however, is beyond the scope of this paper.

[10] The $s + s'$ for sequence tuples $s, s' \in S^k$ denotes positionwise concatenation of the sequences in the tuples.

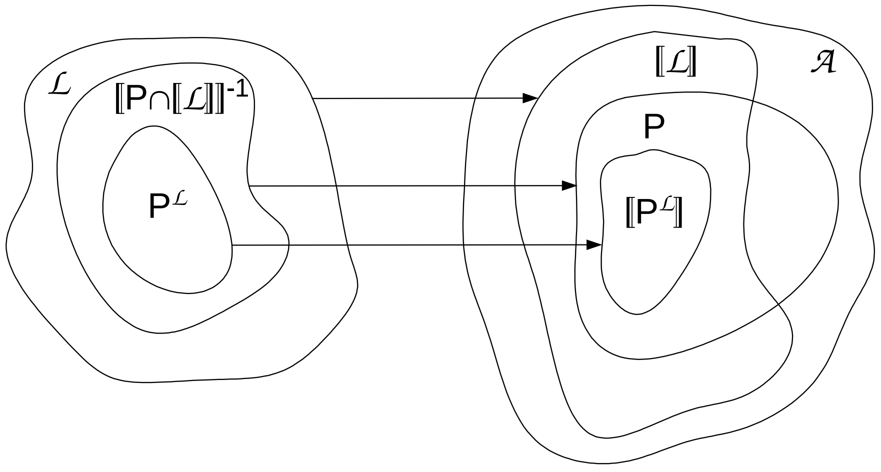
$\mathcal{L}$
$[\![P \cap [\![\mathcal{L}]\!]]\!]^{-1}$
$P^{\mathcal{L}}$
$[\![\mathcal{L}]\!]$
$\mathcal{A}$
P
$[\![P^{\mathcal{L}}]\!]$

Fig. 6. The relation between sets of actors with some property P, sets of actor descriptions with a property $P^{\mathcal{L}}$ approximating P, and their corresponding sets of descriptions and interpretations, respectively.

6. Designing actor and composition languages

In this section we will try to come back to discussing some practical consequences of the formal considerations presented above. Originally, this work came out of the design of an actor language, and of a formal semantics for it, both in the context of the Ptolemy project. Since in Ptolemy actor compositions are also actors themselves, it seemed natural to assume that any reasonably general actor language should therefore be able to express these composite actors.

The next step was to conceive of a model of computation as a program transformation—it generates the description of the composite actor from a model and descriptions of actors used in that model. This is essentially what this paper tries to formalize.

In the context of designing languages for actors and, perhaps more interestingly, composers, one important question is whether the theory we developed surrounding actor composition has any consequences for the design of these languages.

The first thing any composer needs to do when composing a model is to check whether it is well-formed. Well-formedness may depend on many things, such as the syntactical structure of the model, properties of the actors used in the model, and frequently both. For example, in the case of synchronous dataflow, actors must have static production and consumption rates (i.e. at each firing, they must consume the same number of input tokens and produce the same number of output tokens from and at each port)—such actors are often also referred to as *SDF actors*. Furthermore, for a given model, there must be a schedule of actor firings such that after one complete execution of that schedule, the lengths of the buffers between the actors are the same as they were before the execution of the schedule.[11]

Since the composer does not work directly on the actors, but rather on descriptions of them in some language $\mathcal{L}$, it has to determine any property P of an actor based on some other property $P^{\mathcal{L}}$ of actor descriptions. Unfortunately, in practice, for many interesting actor properties P, we can only approximate it by some $P^{\mathcal{L}}$, as shown in Fig 6.

For example, assuming our language is the one introduced in section 2, let us call the property of being an SDF actor SDF, we might choose $SDF^{\mathcal{L}}$ to be the property that all input patterns and all output expressions corresponding to a particular port have the same length in all actions. For example, the following program clearly has property $SDF^{\mathcal{L}}$:

```
**actor** $F$
  $V, N \Longrightarrow Out$ :
  **action** $V : [x, y, z], N : [n] \Longrightarrow Out : [x^n + y^n, z^n]$
```

[11] Cf. [LM87] for more details.

Likewise, the `Multiply` program describes an SDF actor, and so does this one:

```
**actor** $Abs$
  $In \Longrightarrow Out$ :
  **action** $In : [x] \Longrightarrow Out : [x]$
    **guard** $x \geq 0$
  **action** $In : [x] \Longrightarrow Out : [-x]$
    **guard** $x < 0$
```

By contrast, neither the `Split` nor the `Mem` descriptions describe SDF actors.

Clearly, for any property $P^{\mathcal{L}}$ to represent an actor property P, it is important that $[\![P^{\mathcal{L}}]\!] \subset P$, i.e. any program that has this property describes an actor that has property P. In general, however, it will not be the case that $P^{\mathcal{L}}$ completely represents P, i.e. we will not have $[\![P^{\mathcal{L}}]\!] = P$. For example, the following description is semantically equivalent to the description `F` above (assuming all our tokens are non-negative integers), even though it does not have property $SDF^{\mathcal{L}}$:

```
**actor** $FLT$
  $V, N \Longrightarrow Out$ :
  **action** $V : [x, y, z], N : [n] \Longrightarrow Out : [z^n]$
    **guard** $n > 2 \wedge z \neq 0 \wedge x^n + y^n = z^n$
  **action** $V : [x, y, z], N : [n] \Longrightarrow Out : [x^n + y^n, z^n]$
    **guard** $n \leq 2 \vee z = 0 \vee x^n + y^n \neq z^n$
```

Not only does this actor description not have property $SDF^{\mathcal{L}}$, it is clear that it is in general very difficult to show that it describes an SDF actor, in other words there is no other property $SDF_*^{\mathcal{L}}$ such that we could automatically deduce that `FLT` describes an SDF actor with reasonable effort.

Fig. 6 gives an overview of these relations. It also shows that in general, $[\![\mathcal{L}]\!]$ could be a proper subset of $\mathcal{A}$. This is so for some very fundamental reasons concerning the size of $\mathcal{L}$ and $\mathcal{A}$.[12] In addition, depending on the actor language, there may be even stronger limitations to what can be expressed in it—for example, the language introduced in section 2 does not allow to express actors that determine the number of tokens they consume per firing on an input port by, say, a state variable that contains any natural number, or perhaps even a token consumed via some other input port.[13]

From this we can infer two main principles that should guide the design of an actor language. The language should (a) be general enough to allow the definition of a wide variety of actors, and it should (b) provide constructs that allow an appropriate representation of 'interesting' actor properties P by properties $P^{\mathcal{L}}$.

The second point is somewhat difficult, because the notion of 'interesting properties' is by no means a well-defined one. In fact, which properties are important for composing actors depends on the specific composer, i.e. the model of computation. As a consequence, the design of an actor language will impact the kinds of compositions that can be performed on it, and also the difficulty with which they can be implemented. Nonetheless, there are several questions about properties of actors that play a role in many different kinds of compositions, and which any actor description that attempts to support a variety of models of computation should allow to represent. Among others, they include the following:

1. Does the actor have state?
2. Does the actor have only a finite number of states?
3. Is the actor determinate?
4. What do the token rates depend on? State? Presence of tokens on other ports? Token values? Nothing at all?
5. Does the actor have constant token rates?
6. What do output values depend on? Input values? State?

[12] If we assume that there are only countably many actor descriptions, which is of course the case if the language is textual, and that actors allow integer tokens and arbitrary functions from inputs to outputs, it follows that the set of actors is uncountably large, and that therefore not all actors can be represented in the language.

[13] These are only limitations of our toy language—Cal [EJ02] allows actors to express much more sophisticated patterns of token consumption.

A well-designed actor language provides constructs that allow to express these kinds of properties of an actor so that composers may analyze them with reasonable effort.

The impact of this work on possible designs for a language for writing composers, and therefore models of computation, is still very much an open issue. First of all, it seems clear that the composer language will, to an extent, depend on the actor language, because the latter will be the 'data' that the former will operate on, hence a composer language will most likely contain primitives that allow it to construct and manipulate actor descriptions. Some of these are fairly generic, and probably useful in almost all conceivable composers—e.g., assuming we are dealing with a language like the one in section 2, there will probably be facilities for renaming variables consistently in a block of code, for wrapping a block of code into a procedural or functional abstraction, for creating actions with certain input patterns and output expressions computed from patterns and expressions of other actions, and for creating new uniquely named variables.

An important goal in the design of a composer language is to structure the description of composers in a way that makes it possible to prove properties of the composers, or perhaps more realistically, at least allows to gain a much higher confidence in their correctness (which would be defined by their correspondence to some abstractly defined model of computation). Such a composer language will not only support low-level program transformations like the ones described above; instead, it will also provide a richer predefined scaffolding from which composers can be constructed. Such a language might do this at the expense of generality. For example, a composer language for dataflow models of computation might provide ways to define the representation of buffers in the residual state (the β function), as well as how the i and b are defined for any given transition, and of course which transitions are possible in any given state. Even if it does not automatically ensure that these definitions by the composer author actually meet the conditions in Def. 12, it might structure the definition of dataflow models of computation in such a way that it becomes much easier to see that a specific composer is in fact a dataflow composer. Furthermore, for any theorem of the form "if the β, i, b (and possibly others) have property p, any composition has property q", a dataflow composer whose definition makes those things explicit reduces the problem of showing property q for its compositions to showing property p for the parts of its definition.

Finally, much of the discussion about properties of actor compositions relied on the knowledge of the *decomposition*, or at least of some of its properties. It would be very useful if a composer language would also allow the construction of the decomposition—not only for theoretical analysis of its properties, but also for very practical applications such as, e.g., debugging compositions at runtime.

7. Related work

Actor-based models for concurrent systems have a long tradition. The original use of the term in this context is probably due to Hewitt [Hew77], founding a rich area of research. [Agh86, AMST93] In this model, actors are themselves sequential entities, communicating with each other via message passing. In the original model, actors had to know the receiver of a message they intended to send. [Tal96] introduces a more compositional model for actor-based systems, where components are networks of actors that are connected with what effectively amounts to ports, i.e. abstractions for senders and receivers beyond the component boundaries.

As a somewhat related abstraction, Kahn introduced process networks as a model of computation for describing compositions of parallel processes. [Kah74] Processes in this model could formally be interpreted as functions on potentially infinite streams of data. Apart from the fact that these networks guaranteed determinacy under some simple conditions, they had the interesting property that atomic processes and networks of processes behaved identical under composition—a network was itself just a function on streams of data. As a consequence, process networks could be composed hierarchically, making this a *compositional* model of computation. Specialized variants of process networks were developed that allowed more extensive static analysis resulting in more assurances about the system behavior and possibly more efficient implementations—such as e.g. synchronous dataflow [LM87] and cyclo-static dataflow [EBLP94]. See also [Lee94] for an overview.

Dennis introduced a dataflow language that was built on a notion of *firing*, i.e. an atomic step where a component would consume and produce a finite amount of data. [Lee97] showed how some Dennis dataflow networks with firing could be considered as implementing a stream function, thereby effectively unifying Kahn process networks and Dennis dataflow. One remaining problem was the composition of the notion of firing, and the way it interacted with state. [Jan00] provided a more general treatment of state, though it did not address the composition of actor firings.

The bewildering variety of ways in which conceptually concurrent processes/tasks/actors etc. can be composed gave rise to the work reported in [LSV98], which presents a denotational framework in which a large number of so-called *models of computation* can be represented and compared to each other. Models of computation in this context are mathematical models for concurrent systems, that define rules for the interaction of the concurrent, and possibly communicating/interacting parts of a system.

The Ptolemy project [DHK$^+$01] developed a modeling and simulation environment based on a notion of *actor* that included communication via ports and a notion of atomic step called *firing*, similar to Dennis dataflow. It is explicitly open in the sense that it allows the definition of new models of computation. An interesting contribution of Ptolemy is the notion of *domain polymorphism*: the same actor may be used under different models of computation, even among those that have not even been designed when it was written. This is achieved by defining an *abstract semantics*—a semi-formal contract between an actor and any model of computation that allows any actor and any model of computation that adhere to that contract to interoperate. One part of that abstract semantics is the notion of firing, i.e. of an atomic step.

Another important contribution of Ptolemy is a compositional notion of firing. A composite actor, governed by a model of computation, adheres to the same abstract semantics as any atomic actor. As a consequence, composite actors may be embedded into models in the same way as atomic actors. This makes it possible to have different models of computation at different levels of a hierarchical model. [EJL$^+$02] The work reported here can be seen as an attempt to cast the experience with the Ptolemy software into a somewhat less operational framework.

8. Discussion and conclusion

In this paper, we have presented a semantical framework for actor-based modeling and design languages. It is motivated by a desire to describe not only the meaning of actor languages, but also the meaning of different ways of building systems from actors, and how these different modeling languages interact with each other in heterogeneous models.

The key notion explored in this work is that of actor composition realized as a transformation of actor descriptions and a model into an actor description representing the composite actor. The resulting flexibility in composing actors is at the core of the power of the notion of model of computation, but it also poses a major challenge for any formal treatment of this concept, as well as for its implementation. There is, in principle, no limit to what a composer may do with the components in order to compute the composite. However, most 'meaningful' models of computation manipulate actors in some structured way. Discussing a model of computation, and its meaning, is tantamount to discussing just that structure. This work essentially tries to provide a basic vocabulary for these discussions—it develops the theory independent of any specific syntax for describing actors or for building systems, allowing it to be applied to a wide range of actor-based systems. Part of this theory are properties of actor compositions that may be regarded as universal in the sense that every composer must ensure that these properties hold for its compositions. Almost as a side effect, we have thus developed a formal notion of *model of computation*, where these universal properties define fundamental requirements on any model of computation.

One result of this work is that in spite of the abstraction from syntax, and in spite of the very general and seemingly unstructured notion of actor composition, it is still possible to discuss composer properties, to analyze them, and characterize specific classes of composers. Furthermore, it is possible to use the formal techniques developed in this context as guidance for engineering programming languages for actors and composers.

This work can be built upon in many ways. First of all, its actor model, in spite of being sufficiently powerful to serve as a platform for all the work in this paper, can be extended in several ways to better match practical modeling requirements. One obvious extension is the addition of a notion of time, so that some kind of global time can serve as a coordinating factor across actor boundaries. Another possible extension would be the addition of actor-internal concurrency. While these extensions might affect some of the definitions and theorems, it seems likely that the basic form of the theory will remain the same, and so will many fundamental properties and definitions.

On the practical side, we hope to use this theory as a guide for our work on the design and implementation of an actor language [EJ02], and of composer languages. We currently use a general structured document format for representing actors, and a generic transformation infrastructure based on this format for representing actor transformations.[14] It would be highly desirable to have more specialized and expressive ways to write composers, and we hope that this work is a step in this direction.

References

[ABG95] Pascalin Amagbégnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the conference on Programming language design and implementation*, pages 163–173. ACM Press, 1995.

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

[AMST93] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1993.

[Ber00] G. Berry. *The Foundations of Esterel.* MIT Press, 2000.

[BGJ+97] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciano Lavagno, Alberto L. Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems—The POLIS Approach.* Kluwer Academic Publisher, Boston, May 1997.

[Cal] The Caltrop Project. Department EECS, University of California at Berkeley (*http : //www.gigascale.org/caltrop*).

[DHK+01] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, USA, March 2001.

[EBLP94] Marc Engels, Greet Bilson, Rudy Lauwereins, and Jean Peperstrate. Cyclo-static dataflow: Model and implementation. In *28th Asilomar Conference on Circuits, Signals and Systems*, November 1994.

[EJ01] Robert Esser and Jörn W. Janneck. Moses: A tool suite for visual modeling of discrete-event systems. In *Symposia on Human-Centric Computing (HCC '01)*, pages 272–279. IEEE Computer Society, September 2001.

[EJ02] Johan Eker and Jörn W. Janneck. Caltrop—language report. Technical Memorandum UCB/ERL ???/??, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002.

[EJL+02] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software (to appear)*, October 2002.

[GK97] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[GSV02] Gregor Goessler and Alberto Sangiovanni-Vincentelli. Compositional modeling in Metropolis. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proc. EMSOFT'02*, October 2002.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, June 1977.

[Jan00] Jörn W. Janneck. *Syntax and Semantics of Graphs—An approach to the specification of visual notations for discrete event systems.* PhD thesis, ETH Zurich, Computer Engineering and Networks Laboratory, July 2000.

[JE01] Jörn W. Janneck and Robert Esser. A predicate-based approach to defining visual language syntax. In *Symposia on Human-Centric Computing (HCC '01)*, pages 40–47. IEEE Computer Society, September 2001.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress.* North-Holland Publishing Co., 1974.

[Lee94] Edward A. Lee. Dataflow process networks. Memorandum UCB/ERL M94/53, Electronics Reserach Laboratory, July 1994.

[Lee97] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Report UCB/ERL M97/3, EECS, University of California at Berkeley, January 1997.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[LSV98] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[Tal96] C. Talcott. Interaction semantics for components of distributed systems. In *Proceedings of FMOODS*, 1996.

[14] More precisely, we currently use XML/DOM as a format for actors, and XSLT as our language for writing composers (and other actor-manipulating programs, such as code generators). Cf. [Cal] for more details.