# Designing Interfaces to Analyze Composition

Haiyang Zheng and Rachel Zhou
{hyzheng,zhouye}@eecs.berkeley.edu

University of California, Berkeley
Berkeley, CA, 94720, USA

## Abstract

*Component-based design is widely used in embedded software design, where a new component can be generated by composing existing components. However a composition does not always generate a valid component. We define a composition to be valid with respect to a property if the composition preserves that property under certain context. In order to reason about whether a composition is valid, we need to use interfaces to represent the properties we are interested in. In this paper, we explore possible designs of interfaces for this purpose and deploy these interfaces to analyze a few examples with different models of computation.*

## 1 Introduction

Component-based design is widely used to design complex embedded software. The idea is to compose existing components into new components, which can be further composed. By reusing existing components that are well developed and tested, this design technology greatly improves productivity and robustness.

One key requirement to guarantee the reusability and compositionality of components is to ensure that certain properties of the primitive components we are interested in are preserved after composition. We define a composition to be *valid with respect to a property* under a certain *context* if it preserves that property after composition. A context defines the environment where a composition happens, e.g., the model of computation of a model, or the scheduling algorithm used by a simulation engine.

Usually, we are only interested in some properties of a component. We use an interface $I$ to describe those properties and abstract away the other unrelated ones. Obviously, a component can have many different interfaces.

One simple but very common interface for a component is its ports and parameters, denoted as $I_p$. But the component properties this interface talks about is very limited. By associating ports with a data-type system, we get a more informative interface $I_{pt}$. This interface constrains all possible types of data that can be passed through a port. An even more informative interface $I_b$ captures $behaviors$ of a component, which includes not only ports and parameters but also $signals$ going through the ports, where signals satisfy the process defined by this component [7][8]. The last interface we want to mention here is a function abstraction $I_f$, which maps a set of input signals to a set of output signals. This interface provides the most complete information out of the above four interfaces. Usually, it is very complicated.

Depending on what properties of a component we are interested in, we need different interfaces. We define an interface to be *sufficient for a property* if it can correctly expose or carry that property. Further more, we define an interface to be *efficient for a property* if it does not contain more information than necessary. An efficient interface excludes unrelated properties and makes our analysis easier. We define a *good* interface to be both sufficient and efficient for a property. A good interface allows us to reason about whether a composition is valid with respect to that property.

In this paper, we focus on exploring possibly good designs of interfaces to solve some practical composition problems. In section 2, we give an example to illustrate the insufficiency and inefficiency of the interfaces we talked above. In section 3, we give a good interface for discrete-event (DE) models [4] to reason about the preservation of their causality properties after composition. In section 4, we develop more interfaces to analyze compositions of dataflow and process networks models. In the end, we give conclusions and future work.

## 2 An Introduction Example

We begin with a simple hierarchical DE model with a feedback loop shown in Fig. 1. This model has a *CompositeActor*, which is a composition of a *Scale* actor and a

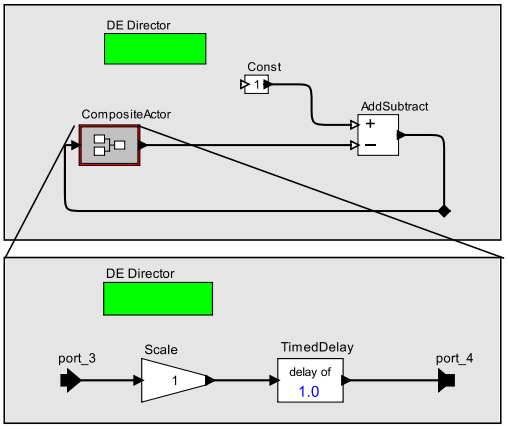


**Figure 1.** A hierarchical discrete-event model with a feedback loop.

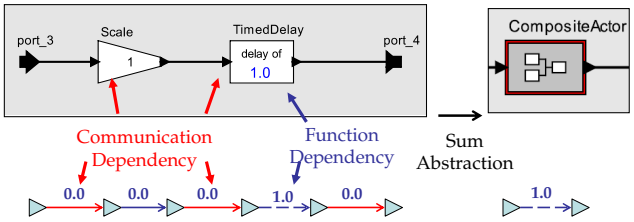


**Figure 2.** Constructing a function dependency of a composite actor from those of atomic actors.

*TimedDelay* actor. The property of this model we are interested in is whether this model has a unique behavior.

If we flatten the CompositeActor, we can easily tell that the system is delta causal because the TimedDelay actor is delta causal. This delta causal property guarantees that this model has a unique behavior [2][7]. In order to draw the same conclusion with composition, we need to tell whether the CompositeActor is a delta causal process. Unfortunately, an interface $I_p$, which consists of only ports and parameters, does not carry such information. Therefore, $I_p$ is not a sufficient interface for us to reason about the uniqueness and existence of the model's behavior.

The reason for the insufficiency of $I_p$ to analyze causality properties is that $I_p$ abstracts away too much information. $I_{pt}$ is not sufficient for the same reason. $I_b$ seems to be a sufficient interface. However, it is very difficult to use in practice because we have to verify the set of all possible signals. $I_f$ is a sufficient interface for our purpose. However, for a rather complex model, $I_f$ itself may be very complicated to analyze. What is more, we argue that $I_f$ is not efficient because what we really need is only one aspect of $I_f$, the causality.

In the next section, we propose a sufficient design of interfaces for causality analysis of DE models. This new interface only captures the causality aspect of $I_f$. Therefore, it is easier to be analyzed.

## 3 An Enhanced Interface

### 3.1 Function Dependency

We need to find an interface that can carry the causality property of an actor. In order to do that, we first define a *function dependency* for an actor. Intuitively, a function dependency of an actor is a data dependency relation that an output port has on an input port of the same actor in a firing. If we treat a function dependency as a function that maps an input to an output, a function dependency has one of the three kinds: *directly affected* (DA), *indirectly affected* (IA), and *not related* (NR). A typical actor that has a DA function dependency is the Scale actor, whose output changes whenever its input changes. A TimedDelay actor (with a fixed non-zero delay: $t$ time units) has an IA function dependency because its output does not change until $t$ time unites after its input changes. An NR function dependency simply says that there is no relation between an input and output.

With these three kinds of function dependencies, we can tell some of the causality property of an actor. To be specific, if an actor has no DA function dependency, it is a *strictly causal* process. Otherwise, the actor is *causal* but not *strictly causal* [1].

With function dependency, we can abstract an *atomic* actor [2] into a directed graph, where nodes are ports and directed edges represent dependencies of destination nodes (output ports) on the source nodes (input ports). In particular, if there is no edge between two nodes, it means these nodes are not related at all (NR). The function dependencies of the TimedDelay actor and the Scale actor are shown as blue edges in Fig. 2. Note that the function dependency of TimedDelay is a dashed arrow (IA) while that of the scale actor is a solid arrow (DA).

We define the directed graph of an actor as an interface with function dependency of that actor. These directed graphs can be further composed with *communication dependencies*, which describe the data dependencies between different actors, to construct the dependency graph of a composition. Fig. 2 shows such a process, where the red arrows indicate communication dependencies. The interface of a composite actor is an *abstraction* of such dependency graph, which captures the dependencies of the output ports of the composite on its input ports. In this example, the

[1] In this paper, all actors by default are causal processes.

[2] An atomic actor is a primitive component that can not be decomposed, whose function dependency is predefined.

composite actor has an IA function dependency.

### 3.2 An Interface with Weighted Function Dependency

According to Banach fixed point theorem [2], strict causality can not completely determine the existence and uniqueness of the behavior of a DE model. Therefore, we need an interface that distinguishes between strict causality and delta causality.

We enhance the interface (function dependency graph) given in the previous section by associating a weight to each edge of the graph, where the weight represents the amount of delay an edge introduces. An example graph is shown in Fig. 2, where the TimedDelay actor introduces a delay of amount 1.0 time unit. Note that the delay of a communication dependency and a DA function dependency is always zero. With a weighted graph, we can calculate the weight (delay) between a pair of input and output ports of a composite actor by summing the weights of all the edges between them.

Now we can distinguish a delta causal actor from a strictly causal actor with the enhanced interface. Given a strictly causal actor, if the weights of all function dependencies are bigger than an $\varepsilon$, where $\varepsilon$ is a positive real number, this actor is delta causal. Otherwise, this actor is just strictly causal. As in the example in Fig. 2, the composite actor is a delta causal process because the weight of its function dependency is 1.0. Consequently, we can conclude the whole model has a unique behavior.

In order to study the causality property of a composition, we only care whether the composition is causal, strictly causal, or delta causal. We are not really interested in the exact quantity of delay from one port to another. Instead, we care whether such delay is zero, greater than zero, or greater than some positive real number $\varepsilon$. We represent these conditions with three elements $0$, $>$, and $>_\varepsilon$ respectively.

Then we can simplify our interface by representing the weight on an edge as one of these three elements. We define the summation operation $\oplus$ performed on these elements in the table below. Here is an example of how to read the table. $> \oplus >_\varepsilon = >_\varepsilon$ means that if a strictly causal function dependency is cascaded with a delta causal function dependency, the resulting function dependency is delta causal.

| $\oplus$ | $0$ | $>$ | $>_\varepsilon$ |
|---|---|---|---|
| $0$ | $0$ | $>$ | $>_\varepsilon$ |
| $>$ | $>$ | $>$ | $>_\varepsilon$ |
| $>_\varepsilon$ | $>_\varepsilon$ | $>_\varepsilon$ | $>_\varepsilon$ |

## 4 More Enhanced Interfaces

### 4.1 Enhanced Interfaces on Dataflow Models

In Ptolemy II [1], dataflow models are scheduled hierarchically in a bottom-up fashion. An opaque composite actor is treated the same as an atomic dataflow actor. The interface of a composite actor is its *rate signatures*, which are the number of tokens consumed and produced by the composite actor in one iteration [9]. However, rate signatures only capture the consistency property of dataflow models [5], but talk little about deadlock.

Let us revisit the example in Fig. 1 by replacing the TimedDelay actor with a *SampleDelay* actor and changing DE directors to dataflow directors, e.g., synchronous dataflow (SDF) directors [5]. If the hierarchy does not exist, the model is deadlock free since the SampleDelay actor produces an initial token to break deadlock and start execution. With composition, we need a good interface to expose the information about initial tokens.

Again, we introduce a weighted interface for dataflow models to capture information about initial tokens. Like in Section 3, nodes represent ports and directed edges represent function and communication dependencies. One key difference here is we associate a weight to each node, rather than to each edge. The weight is a non-negative integer that represents the number of initial tokens available at that node (port). Another key difference is that the directed edge represents a function that maps the weight of its source node to that of the destination node. For communication dependencies, the weight of destination node is the same as that of its source node, i.e., the function is an identity function. Fig. 3 shows the functions defined for different actors. For example, the SampleDelay actor with parameter $\{0\}$ produces one more token than its input in its initialization phase. Afterwards, it produces the same number of tokens that it receives. For an actor with consumption rate $m$ and production rate $n$, and given $k$ initial inputs, the actor can produce $\lfloor k/m \rfloor \times n$ tokens.

The interface of a dataflow composite actor is an ab-

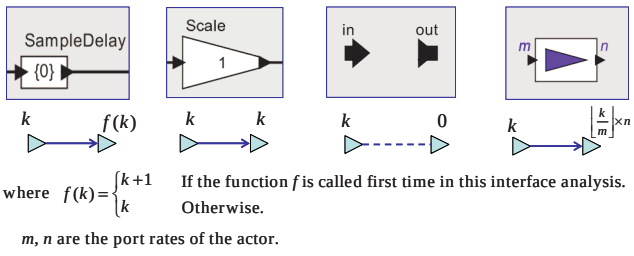


**Figure 3.** Weighted interfaces for actors in dataflow models.

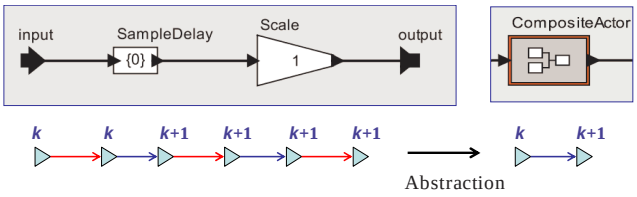


**Figure 4.** Constructing an interface abstraction of a dataflow composite actor from atomic actors..

stracted function that maps the number of initial tokens from the composite input ports to those of its output ports. An example is shown in Fig. 4. The abstracted function maps $k$ initial tokens into $k + 1$ tokens, where $k$ can be any non-negative integer. This interface captures the information that the composite can produce one initial token without any input tokens (when $k = 0$) to break the feedback loop and resolve the deadlock.

### 4.2 Detecting Disconnected Processes

With the function dependency interface introduced in Section 3.1, we can detect disconnected submodels in a model and therefore solve the artificial local deadlock problem [3] in for process network (PN) models that run with Parks' algorithm [6]. According to Parks' algorithm, if a global artificial deadlock occurs, the model stops executing and increases the buffer size of one blocking write channel to proceed execution. However, if a process network model has two disconnected processes, and if one is artificially deadlock but the other one is still running, Parks' algorithm doesn't increase the buffer size to solve this (local) deadlock.

Our solution is to separate disconnected graphs into several submodels and assign each one with a local PN director. This director is responsible to manage its local buffer size to resolve local artificial deadlock. We have implemented this decomposition procedure in Ptolemy II.

## 5 Conclusion and Future Work

In this paper, we introduced an approach from the interface abstraction aspect to analyze composition. We developed several interfaces and applied them to several models with different models of computation.

Interface analysis can also help to detect disconnected subgraphs in a model and solve the artificial local deadlock problem of Parks' algorithm.

For future work, we plan to study and develop more useful interfaces for compositionality analysis.

## References

[1] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java. Technical Report Technical Memorandum UCB/ERL M04/17, University of California, June 24 2004.

[2] V. Bryant. *Metric Spaces*. Cambridge University Press, 1985.

[3] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *European Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, pages 319–334. Springer, 2003.

[4] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[5] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 1987.

[6] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[7] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998.

[8] E. A. Lee and Y. Xiong. System-level types for component-based design. In *First Workshop on Embedded Software, EMSOFT 2001*, volume LNCS 2211, Lake Tahoe, CA, 2001. Springer-Verlag.

[9] Y. Zhou. Communication systems modeling in Ptolemy II. Masters Thesis Technical Memorandum No. UCB/ERL M03/53, University of California, Berkeley, December 18 2003.