Preserving Real Concurrency

James Ivers and Kurt Wallnau

Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. We are concerned with reasoning about the behavior of assemblies of components from models of components and their patterns of interaction. Behavioral models of assemblies are "composed" from behavioral models of components; the (composed) assembly model is then mapped to one or more reasoning frameworks, each suitable for a particular kind of analysis. Information relevant to each reasoning framework must be faithfully rendered in the assembly model, and it must be preserved under interpretation to the reasoning framework. Our concern in this paper is the faithful rendering of concurrency. Our approach makes use of information provided by components and extracted from static assembly topologies to faithfully model real concurrency. The result is more effective analysis.

1 Introduction

Our work is concerned with predicting the behavior of assemblies of components from the specifications of components and from their patterns of interaction. Behavioral models of assemblies are "composed" from behavioral models of components; the (composed) assembly model is then mapped to (interpreted in) one or more reasoning frameworks, each of which is based on its own computational model, and each of which is used to predict some directly or indirectly observable behavior of an executing assembly. Information relevant to each reasoning framework must be faithfully rendered in the (composed) assembly model, and it must be preserved under interpretation to the reasoning framework.

Our concern in this paper is with the correct rendering of implementation concurrency in assembly models that are mapped to temporal logic model checkers [1]. In particular, we are concerned that the composed model neither over- nor under-approximates concurrency. Over- and under-approximation are the inclusion of more or fewer processes (respectively) in a model than exist in the corresponding implementation. Over-approximation leads to spurious counterexamples that take time and resources to eliminate and contributes to statespace explosion, as it results in additional interleavings that must be evaluated. Under-approximation is more insidious, since it leads to missed counterexamples and, potentially, to runtime errors. Composing models that are faithful to real concurrency is not trivial for the kinds of reactive systems that we are addressing. These systems are concurrent and distributed but not massively so; in particular, components may exhibit internal concurrency, may execute behavior on the caller's thread of control, and may (and typically) do both.

In this paper we outline our approach to composing assembly models that are faithful to real concurrency. We require that some aspects of the internal concurrency structure of components be exposed; we specify these aspects through component *reactions*. Information gleaned from a static topology of an assembly allows us to compose reactions in a way that is faithful to real concurrency.

The rest of this paper is structured as follows. Section 2 provides some background on our approach to predictable assembly and states our composition problem in a bit more detail. Section 3 outlines our component specification and assembly composition approach. Section 4 briefly touches on some related work, and Section 5 summarizes.

2 Background

Our approach to predictable assembly is to construct prediction-enabled component technologies (PECTs). The central idea of a PECT is that a construction model of a system, specified in terms of an assembly of components [2], can be mapped (via syntactic interpretation) to one or more analytic models, each corresponding to a particular computational model and reasoning technique. A construction model describes a component-based system in terms of these characteristics:

- A *component* has interfaces, specified as *pins*, that accept stimulus from its environment (*sink* pins), and can initiate stimulus on its environment (*source* pins). Pins can produce and consume data, and are specified with a signature similar to conventional APIs. Pins are specified as supporting either synchronous (call/return) or asynchronous (message based) interaction.
- An *assembly* is a set of components composed in a particular *runtime environment*. The runtime environment provides connectors to connect the source pins of one component to the sink pins of another. Connected pins must be conformant: their signatures and interaction modes must match. The assembly must also obey other connector-imposed constraints (1:1, 1:N, etc.).

Because components are composed with different combinations of other components to form different assemblies, their exact behavior is not always knowable based only on their own specifications. For example, assume that a component has two sink pins, and that the behavior of each modifies some shared (internal) variable. This component may behave differently depending on the context in which it is composed. There is no possibility violating mutual exclusion if both pins execute on the same thread. However, if each pin can execute on a different thread, then mutual exclusion violations are possible unless the calling threads are coordinated. Each of these cases calls for a different composed model of behavior, which can only be determined when the component's context is known.

3 Approach

Our solution to systematically achieving correct composition with respect to order dependent properties of concurrent systems relies on using the following process, the steps of which are elaborated in the following sections:

- Allocate component behavior to potentially concurrent units called reactions.
- Model each reaction.
- Compose reactions in a way that preserves implementation concurrency.

3.1 Allocate Behavior to Reactions

We begin by examining a component's implementation, which has some intrinsic concurrency and some undetermined concurrency. Intrinsic concurrency comes from threads created and managed by the component. Certain behavior of the component only executes in these threads. Undetermined concurrency comes from units of behavior that are not allocated to threads by the component, such as function calls. The real concurrency of such units of behavior will depend on how the component is composed with other components in its environment.

When modeling a component in a reactive system, we focus on how it responds to stimulation of its sink pins. A natural starting point would be to produce one model for each sink pin that shows how the component responds when stimulated on that sink. However, this approach could model more concurrency than is implemented in the component. Instead, we use our understanding of the component's potential concurrency to allocate the behavior for handling each sink pin into potentially concurrent units called reactions.

A *reaction* is a model of a collection of behavior that always executes within the same thread of control. A reaction describes the relationship between a collection of sink and source pins by defining how the source pins are stimulated in response to stimulations of the sink pins. For example, a particular reaction could model a thread of a component that retrieves messages from a queue (sink pin stimuli), performs some computation based on the type of message received, and sends messages (source pin stimuli) based on the results of the computation.

Behavior is allocated to reactions following prescribed rules based on a component's intrinsic and undetermined concurrency:

- The behavior of each sink pin is allocated to exactly one reaction.
- All sink pins that are handled by the same thread of the component must be allocated to the same reaction. Each such reaction is called a threaded reaction and represents a unit of intrinsic concurrency in the component.
- Each sink pin that is not handled by a thread of the component is allocated to a reaction of its own. Each such reaction is called a non-threaded reaction and represents a unit of undetermined concurrency in the component.

Recall that a reaction is a *potentially* concurrent unit. Allocation of each sink pin of undetermined concurrency to a separate reaction is a pessimistic

decision. While two such pins may always be stimulated by the same thread in a particular assembly and so could be allocated to a single reaction, this cannot be determined from the implementation of the component. Such decisions are made later in our process, when the context is known and reactions are composed.

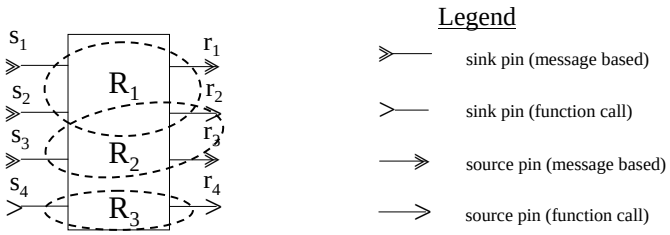

Fig. 1. Allocation of component behavior to reactions.

Figure 1 is a graphical representation of a component and the allocation of its behaviors to reactions. The dashed ovals represent an approximation of how the behavior of the component is allocated to reactions R_1 through R_3.

This component has two threads (not shown graphically). The first handles sinks s_1 and s_2 and may stimulate sources r_1 and r_2; therefore, all behavior associated with these two sinks, including the circumstances under which this thread will stimulate these sources, is gathered into a single threaded reaction, R_1. The second thread handles only sink s_3 and may stimulate sources r_2 and r_3; all behavior associated with s_3 is allocated to the threaded reaction R_2.

The last sink, s_4, is a bit different. It is not handled by one of the component's threads; instead, it executes on the thread(s) of any clients interacting with that sink. The implementation handling that pin is still part of the component (e.g., a function call exported by the component), but the execution of the implementation handling that pin is not on one of the component's threads. Consequently, the behavior associated with this sink pin is allocated to its own non-threaded reaction, R_3, and its true concurrency is not known until the component is composed with other components.

3.2 Model Reactions

The sequential processing performed in each reaction is modeled using a state machine based description. We use a variation of UML statecharts extended with an action language based on a subset of C, much like the action language of xUML [3].[1]

The level of detail found in each reaction varies. Due to model checking limitations (notably the statespace explosion problem), reactions are usually abstractions of the behaviors they represent. As our goal is an understanding of

[1] Modeling reactions with statecharts is a modification to earlier work in which reactions were modeled in CSP [4]. We adopted statecharts because CSP was found to be too difficult for expected users.

the potential sequences of pin stimulations and states during system execution, we concentrate on the control structures within reactions, and how they influence how a component interacts with its environment via its pins.

Typically, each reaction begins in an initial state in which it will accept input (stimulation) on any of its sink pins. What happens next is dependent on the component, but typically each reaction reaches a point where it concludes the processing for that sink pin stimulation and returns to a state in which it will again accept input on its sink pins.

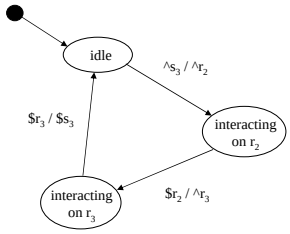

Fig. 2. Statechart for reaction R_2.

Figure 2 shows a reaction model for R_2 from Figure 1. The behavior of this reaction is rather simple. The reaction begins in the *idle* state, waiting for its sink pin to be stimulated ($\hat{}s_3$ is the event that represents a stimulation of s_3). Stimulation of s_3 initiates an interaction, during which the reaction in turns stimulates source pins r_2 and r_3. However, between stimulating r_2 and r_3, the reaction waits for interaction on r_2 to complete by waiting for the $\$r_2$ event, which indicates completion of the interaction on r_2. After stimulating r_2 and r_3, the reaction concludes the interaction on s_3 by generating the $\$s_3$ event and returns to the *idle* state where it waits for the next sink pin stimulation.

3.3 Compose Reactions

When components are composed in a particular topology, we have sufficient context to correctly model the real concurrency of a concrete system. To produce this model, we compose reactions using the following rules:

1. Recursively eliminate all reactions that are not used in the assembly. That is, if no sink pins of the reaction are connected to source pins of the environment or other components that are used in the assembly, then the reaction will never be stimulated, and does not need to be included in the composition.
2. Determine which reactions stimulate each non-threaded reaction. A reaction stimulates another if one or more of its source pins are connected to one or more of the other reaction's sink pins.
   (a) For each reaction that stimulates a non-threaded reaction, make a copy of the non-threaded reaction and compose it sequentially with the stimulating reaction. The result is that each stimulating reaction grows larger by

incorporating the behavior of the non-threaded reaction. As part of this composition, any source pins stimulated by the non-threaded reaction should now be stimulated by the reaction with which it is composed.

(b) When finished, eliminate the original non-threaded reaction.

3. Combine the remaining reactions, each of which corresponds to a thread in the implementation, using parallel composition.
4. Add statecharts defining the interaction (connector) semantics used in the assembly. For example, two components communicating via message passing would use statecharts defining the semantics of message passing in their runtime environment (e.g., a FIFO message queue that blocks when full).

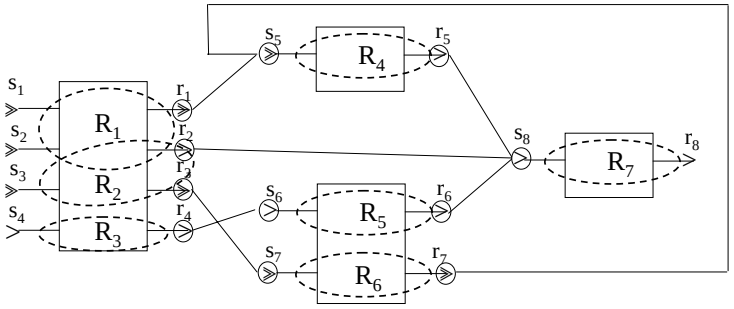

Fig. 3. An assembly of components with component behavior allocated to reactions.

Figure 3 shows a sample assembly and the allocation of behavior to reactions. Applying the above composition rules, we get the following results at each step:

- No reactions are eliminated in step 1, as we assert (but do not show) that each sink pin of the left-most component is connected to the environment.
- The three non-threaded reactions in this example are R_3, R_5, and R_7. Rule 2 is applied to each of these reactions in turn.
  - R_3 is stimulated only by the environment. Therefore a copy of R_3 is sequentially composed with the environment. R_3 is then eliminated.
  - R_5 was stimulated only by R_3. However, after the above step, R_3 was eliminated, and R_5 is now stimulated by the environment due to the sequential composition of R_3 with the environment. Consequently, a copy of R_5 is sequentially composed with the environment, and R_5 is then eliminated.
  - R_7 is now stimulated only by R_1, R_2, R_4, and the environment (via R_5 and the above step). A copy of R_7 is sequentially composed with each of these models, and R_7 is then eliminated.
- After the preceding steps (and ignoring the environment, which is a separate topic), we are left with only four reactions–R_1, R_2, R_4, and R_6. These reactions are now composed in parallel, and each now correctly corresponds to a real thread of execution.

– Finally, the resulting model is composed with models of the connectors, which supply the correct interaction semantics for each type of communication used in the system (e.g., function calls vs. message queues).

4 Related Work

There has been some interest in composing heterogeneous model fragments, e.g., various diagram types in UML [5]. Our concern is composing homogeneous fragments, e.g., executable UML statecharts with the particular end-goal of using the composed models as input to multiple reasoning frameworks. Liang et. al. adopt model composition of Petri nets, and likewise preserve information about real concurrency, although their interest is restricted to RMA schedulability and therefore leads to a coarser treatment of concurrency [6]. Sora et. al. also conclude that the internal flow structure of components must be made explicit for faithful composition, but concurrency is not addressed per se [7]. There are numerous examples of using process algebras to define the semantics of composition languages [8, 9]; however, these efforts adopt simplifying assumptions about real concurrency such that a faithful rendering of concurrency is not a concern. While we have explained how we preserve real concurrency during model composition, we have not addressed a related problem–preserving real concurrency during the interpretation to the input language of a model checker. Work in faithful model translation such as that in [10] is expected to be of great use.

5 Conclusion

Reasoning about the concurrent behavior of arbitrary assemblies, and reasoning from the perspective of multiple computational models, imposes new requirements on the techniques used to specify and compose behavioral models. This is true whether our concern is with e.g., model checking, as in this paper, or with fault tolerance, reliability, or timing analysis.

The approach outlined has drawbacks, and several open questions remain. One drawback is that the explicit modeling of connection mechanisms introduces artificial concurrency under our current interpretations to model checkers. We minimize excess interleavings by combining connection models where appropriate (e.g., when involving shared resources such as message queues), but a more accurate approach may yet be defined. Pragmatically, it is not clear how readily end-users will adopt this approach to specifying potential component concurrency via reactions.

However, there are clearly benefits to this type of approach. By preserving the actual concurrency of a system in its model, we gain confidence that concurrency errors will not be missed and that counter-examples that are reported are indeed relevant (because they, by definition, represent interleavings that are possible in the implementation). We may even open up an opportunity to exploit knowledge of the implementation's scheduling policy. For example, if we know that a thread is only pre-emptible at certain points or that a lower priority

thread can never pre-empt a higher priority thread, this knowledge can be used to eliminate classes of model process interleavings that are not possible in the corresponding implementation.

A last point worth noting is that our emphasis on model composition reflects the distinction between compositional reasoning and reasoning about compositions. The benefits of compositional (and the stronger modular) reasoning are simple: divide and conquer. As noted elsewhere, however [2], the criteria that must be satisfied for compositionality and modularity of reasoning are often too strong to achieve in practical settings. It is in these circumstances that compositional modeling becomes necessary. Nonetheless, an important element of predictable assembly is, and will continue to be, expanding the frontiers of compositional and modular reasoning.

References

1. Edmund M. Clarke, J., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
2. Wallnau, K.C.: Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Sofware Engineering Institute, Carnegie Mellon University (2003)
3. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison Wesley (2002)
4. Ivers, J., Sinha, N., Wallnau, K.: A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, Sofware Engineering Institute, Carnegie Mellon University (2002)
5. van der Straeten, R.: Semantic Links and Co-evolution in Object-oriented Software Development. In: Proceedings ASE 2002. 17th IEEE International Conference on Automated Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society (2002)
6. Liang, P., Arevalo, G., Ducasse, S., Lanza, M., Schaerli, N., Wuyts, R., Nierstrasz, O.: Applying RMA for Scheduling Field Device Components. In: ECOOP 2002 Workshop Reader. (2002)
7. Sora, I., Verbaeten, P., Berbers, Y.: A Description Language for Composable Components. In: Proceedings of Fundamental Approaches to Software Engineering 6th International Conference, FASE 2003. Number 2621 in LNCS, Warsaw, Poland, Springer-Verlag (2003)
8. Giannakopoulou, D.: Model Checking for Concurrent Software Architectures. PhD thesis, Imperial College of Science, Technology, and Medicine, University of London, England (1999)
9. Achermann, F., Lumpe, M., Schneider, J., Nierstrasz, O.: Piccola – a Small Composition Language. In Bowman, H., Derrick, J., eds.: Formal Methods for Distributed Processing–A Survey of Object-Oriented Approaches. Cambridge University Press, Cambridge, UK (2002) 403–426
10. Katz, S., Grumberg, O.: A Framework for Translating Models and Specifications. In: Proceedings of IFM2002 (International Conference on Integrated Formal Methods). Number 2335 in LNCS, Springer-Verlag (2002) 145–164