4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction. Toronto, Ontario, Canada, May 2001.

Verifying Component-Based Collaboration Designs

Kathi Fisler[1], Shriram Krishnamurthi[2], and Don Batory[3]
1: Department of Computer Science, Worcester Polytechnic Institute; kfisler@cs.wpi.edu
2: Department of Computer Science, Brown University; sk@cs.brown.edu
3: Department of Computer Science, University of Texas at Austin; dsb@cs.utexas.edu

1 Introduction

Collaboration-based or layered design provides an architecture for defining software systems. In this architecture, systems are defined as a composition of layers, where each layer (collaboration) defines a feature and the roles that each actor in the system plays in the feature. Collaboration-based designs have the advantage that features are easily added to or deleted from the system; simply changing the composition of collaborations that defines the system changes the features that the system implements. Collaboration-based architectures are receiving increasing recognition as providing a flexible and scalable approach to large-scale system design, particularly for product-line architectures.

Scalable, collaboration-based designs are fundamentally component-based [6, 9]. Collaborations share the features of components [5, 7, 11], at least in principle: they are black boxes that can be combined through external linkage; their composition can be treated hierarchically; and they may be instantiated multiple times when building a system (the latter situation could arise if the same feature is added to different parts of the system at different times). This synergy between components and collaborations provides a practical and important class of problems to drive and evaluate research into components. We are interested in formal verification techniques that support compositional reasoning about component-based, collaboration-based designs. In this paper, we outline a vision of analysis techniques for this design domain and argue that most conventional compositional verification techniques, such as are found in the model checking literature, are poorly suited for this domain.

2 Collaboration-Based Design

Programs consist of *actors* and the *roles* that they play. Collaboration-based designs structure software around *collaborations* (layers) that define how each actor in the system contributions to a given feature or task. Consider a system for managing library circulation information. The system contains two classes of actors: books and patrons, both organized into databases. Circulation operations, such as checking in books or placing them on hold, involve interaction between a patron and a book. A conventional, non-collaborative design would implement one module for books and another for patrons. Each module would contain the code for performing circulation operations relative to each actor. A collaboration-based system, in contrast, would group the information related to each operation into a single component. Composing the components for each desired operation builds the complete software system. Figure 1 shows an example of a library system composed with a collaboration for handling lost library books. Figure 2 shows a general collaboration-based design in contrast to a conventional design (where each state machine corresponds to a single actor). Collaboration-based designs have been used to implement many substantial systems, including an artillery management system [1], a programming environment [6], and extensible reasoning tools [4, 10].

3 A Model of Collaboration-Based Design

To foster a discussion of verification for collaboration-based designs, we provide a model for collaborative designs and discuss the verification challenges within that model. Concretely, we view a design as a set of classes, roughly one per actor in the system. A collaboration consists of a set of class extensions (*mixins*) for the actor classes. The mixins in a collaboration relate to a common task (feature) in the overall system. This definition allows actor classes

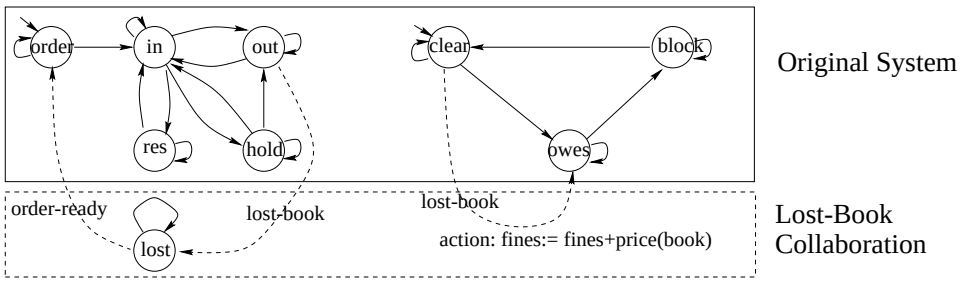

Figure 1: Example of a collaborative design for a library circulation system. The state machines capture books (left) and patrons (right). The dashed box encloses a collaboration which extends the system with functionality for handling lost books.

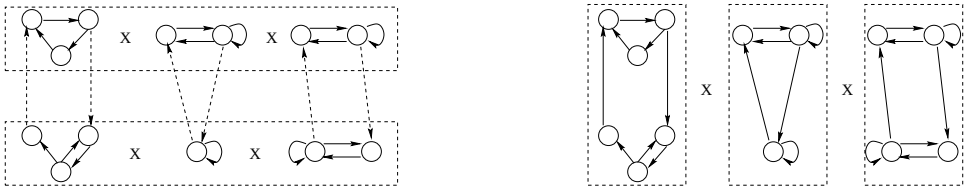

Figure 2: Two views of a collaborative design: sequential composition of layers (left) versus parallel composition of extended actors (right).

and mixins of arbitrary complexity. To make the problem of verification more tractable, we assume each actor class can be described as a state machine, and each mixin extends an existing state machine by adding new nodes and edges. Composition of a layer with an existing system simultaneously refines the state machines of several actors. More specifically, each state machine refinement is defined by a set of entrance states (with a similar match-up of states at exit from the layer). Eventually we would need a way to verify that the state machine is a sound model of the code. Researchers are working on constructing such abstractions automatically, and such models are already available for many systems [2, 3].

Figure 1 presents a simple example of our collaboration-design model on a library circulation system. Books are in one of the states $\{order, in, out, res(erve), hold\}$; patrons are in one of the states $\{clear, owes, block(ed)\}$ (corresponding to levels of fines on a patron's account). Labels on the transitions are omitted here, but support operations such as checking books out and putting books on hold. A later extension to the system adds facilities for handing lost books. This extension involves a new state and path in the book machine (to register a book as lost and possibly order it again) and a new path in the patron machine (to add fines). The two machine extensions form a collaboration. Composing the Lost-Book collaboration with the original system through the dashed edges yields the new library system.

Given a set of library collaborations, we might wish to prove properties about the behavior of a composed library system. For example, we might wish to prove that once a book is reported as lost, the account of the patron who lost the book is assessed fees to replace the book. We would like to reason about such properties at the level of the collaborations, rather than at the level of the entire system. This is largely a tractability concern: full, realistic designs are generally too large to verify naïvely. Verification research looks for ways to reduce the portion of the design that must the analyzed to prove a given property. Collaborations restrict these portions naturally: the lost-book collaboration contains the computation related to lost-book properties. Better still, the lost-book collaboration has fewer states than the full system, so verification should be more tractable on the collaboration level. By abstracting the contents of each collaboration as a collection of state machines, we can investigate the applicability of conventional verification techniques. Model checking, in particular, offers a variety of techniques for specifying systems and properties, and for verifying that systems satisfy properties; this technique has been especially popular in verifying hardware systems. We therefore consider the defining questions in the context of this abstraction.

3.1 What Properties Do We Want to Prove?

We are primarily interested in verifying behavioral properties, rather than performance properties. Focusing on behavioral properties allows us to leverage existing specification logics. These logics can capture a variety of statements about systems including safety (invariants) and liveness (progress) properties.

For collaborative designs, a programmer might ask two natural questions of new collaborations:

1. Does the new collaboration break (global) properties of the existing system?
2. Does the existing system invalidate (local) properties of the collaboration?

In our library example, the Lost-Book collaboration should preserve the property that blocked patrons may not charge books. The original system should preserve the property that losing a book increases the fines that a patron owes. The Lost-Book collaboration would, however, break an existing property that patrons are assessed fines only if they have overdue books. Characterizations of which properties are preserved under extension should derive from similar work in the verification community. The substantial challenge lies in knowing what information to include in the interface of a collaboration to support such reasoning. The remaining sections discuss aspects of this challenge.

3.2 What Compositional Reasoning Techniques are Available?

Compositional verification of modular designs is an area of active research in the model-checking community. Conventional approaches to this problem assume that a system M is composed of modules M_1 and M_2, executing in parallel. To prove a property P of M, one decomposes P into properties P_1 and P_2 such that P_1 (P_2) can be proven of M_1 (M_2), possibly under some assumptions regarding M_1's (M_2's) environment. Combining P_1, P_2, and the environmental constraints in a particular way yields a proof that M satisfies P.

This approach does not naturally apply to the components arising from collaboration-based design. Collaborations extend existing machines rather than operate in parallel with them. The very nature of the extension, which adds paths to an existing design, implies that the collaboration will execute sequentially, not in parallel, with the original design. Most compositional verification theories embody an assumption of parallel composition because they require that composition will never add behaviors to a design. As adding behaviors is the entire goal behind collaboration-based design, most existing compositional verification techniques will not apply in this setting.

We have developed an initial theory of collaboration-based verification. For trivial collaborations involving a single actor, our approach is similar to Laster and Grumberg's [8] work on sequential composition, which was developed independently and has been used for reasoning about hierarchical designs such as those arising from StateCharts. Our full context differs from these works in three key ways. First, Laster and Grumberg attempt to decompose a design into sequential fragments; our fragments arise naturally from the collaborative design architecture. Second, existing work assumes complete, closed systems, rather than systems that will be built (possibly dynamically) from black-box components. Both issues are fundamental in collaboration-based design. These issues also raise substantial questions about component interfaces for compositional verification; closed-world approaches can ignore this question.

The third distinction points to the heart of the technical challenge in this problem: collaborations involve multiple sequential compositions (the mixins) operating in parallel (the whole collaboration). Our goal is to reason about collaboration composition sequentially, even though the overall (extended) actors run in parallel. Figure 2 illustrates the problem at hand: while we may think of collaborations as producing the parallel composition of extended systems shown on the right, we wish to verify the collaborations via sequential composition as shown on the left. Verifying within the sequential view is preferable because collaborations naturally isolate the parts of systems that are relevant to particular properties; this task is extremely difficult under parallel composition. Thus, while we are interested in the same goals of *assume-guarantee reasoning* as found in the modular verification community, this project will need new theories of reasoning about extensions and compositions within collaborative design.

3.3 What Internal Details About Components are Needed?

Ideally, collaboration- and system-interfaces should provide sufficient information to verify large classes of properties without access to a component's internal details. Section 3.6 discusses the interfaces that we envision to support such reasoning. There are at least two circumstances, however, in which access to the entire state machines from each layer may be needed:

- When verifying a new property of the system for which the interface properties are insufficient.
- When module interfaces contain no property-oriented specifications.

The first problem does not apply to lightweight abstractions such as traditional types. However, as we ask more sophisticated questions of a collaboration, we must inevitably expose more of the functionality of the implementation. Since model checking properties ask extremely detailed questions about an implementation, exposing the implementation sometimes becomes unavoidable. The second problem can be addressed by decorating interfaces with property provisions and requirements (see Section 3.6).

In our library system, for example, we might decorate the Lost-Book collaboration with a property that processing a book as lost does not affect the status of other books checked out to the same patron. This information would support a proof that the extended system properly maintains invariants between the book and patron databases with regards to checked-out materials. Similarly, we might annotate the original system with a property that only overdue books increase fines; this information would support a new property about the causes of fines in the new system being limited to overdue and lost books.

We believe that experience verifying collaboration-based designs will yield results about classes of interface properties that are most useful in practice (such as those about which operations leave which attributes intact). We see identifying these classes of properties as one of the short-term challenges for research into compositional verification of collaboration-based designs.

3.4 What Can We Prove Without the Component's Context?

The properties that a collaboration's implementation must satisfy are largely independent of its deployment context. These usually state either consistency or inevitability requirements, and reflect invariants that the underlying program depends on. Therefore, we can state interesting and rich properties of each component independent of its use.

This situation is somewhat different than that in most compositional verification work, which requires substantial environmental assumptions. The nature of collaborations should reduce the complexity of these assumptions, since collaborations encapsulate individual and largely orthogonal features. Operationally, collaborations attach to specific states of the existing state machines, and do not interact much with other states in the existing machines; therefore, the interaction between the new component and the rest of the system is limited. This requires much less contextual information, which traditionally reflects communication between components operating in parallel. This difference is what makes us believe that compositional verification on collaborations may be far more effective than previous similar efforts on parallel systems.

3.5 How to Measure these Properties and with What Precision?

We are interested in behavioral properties, such as are commonly measured through some combination of model checking, theorem proving, or static analysis. Our analyses will be sound with respect to the state machine representations of the design; the state machines may be slightly inaccurate with respect to the low-level code, as discussed in Section 3.

3.6 How Do Components Make Necessary Information Available?

The library example in Figure 1 motivates our intended component interface. The extension layer contains two state machine fragments. Each fragment connects to a corresponding state machine in the original system by adding edges between its start and finish states and states in the original system. The book machine extension, for example, connects to the pair of states $\{out, order\}$. The interface of the original system must specify which pairs of states are valid source and target states for extensions to each state machine. This model captures extensions in the actual collaborative designs that we have studied.

For each state appearing in an interface pair, the interface must also publish a set of formulas that is true at that state; in most cases, these formulas will be automatically derived from the user-specified properties that have already been proven (and should be preserved) of the original system (we have an algorithm for this task). Publishing these formulas is essential to our approach to compositional verification. The interface of a layer states the properties that are true of that layer and that should continue to hold after the layer is added to a system. The Lost-Book layer, for

example, might include the property that a book, once lost, is not checked out to any patron. One research problem is to determine how large these interfaces must be.

4 Conclusion

Collaboration-based designs represent a class of component-based systems that inspire a particular vision of modular verification. Each component in such a design represents a single design feature or operation. The boundaries of these components align naturally with the sorts of properties which are verified using model checking. As developing properties that align with component boundaries is usually one of the main challenges in using composition model checking, we believe collaboration-based design provides a natural framework for exploring component-based verification strategies. This paper has outlined our vision of component-based verification for collaboration-based designs and some of the avenues we intend to explore to achieve this vision.

References

[1] Batory, D., C. Johnson, B. MacDonald and D. von Heeder. FSATS: An extensible C4I simulator for army fire support. In *Workshop on Product Lines for Command-and-Control Ground Systems at the First International Software Product Line Conference (SPLC1)*, August 2000.

[2] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zheng. Bandera : Extracting finite-state models from java source code. In *International Conference on Software Engineering*, 2000.

[3] Dwyer, M. B. and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report UM-CS-1999-052, University of Massachusetts, Computer Science Department, August 1999.

[4] Fisler, K., S. Krishnamurthi and K. E. Gray. Implementing extensible theorem provers. In *International Conference on Theorem Proving in Higher-Order Logic: Emerging Trends*, Research Report, INRIA Sophia Antipolis, September 1999.

[5] Flatt, M. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, 1999.

[6] Flatt, M., R. B. Findler, S. Krishnamurthi and M. Felleisen. Programming languages as operating systems (*or*, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, September 1999.

[7] Heineman, G. T. and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[8] Laster, K. and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[9] Smaragdakis, Y. and D. Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, 1998.

[10] Stirewalt, K. and L. Dillon. A component-based approach to building formal-analysis tools. In *International Conference on Software Engineering*, 2001.

[11] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.