Modular Verification of Collaboration-Based Software Designs

.

Kathi Fisler
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA, 01609 USA
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Computer Science Department
Brown University
Providence, RI, 02912 USA
sk@cs.brown.edu

ABSTRACT

Most existing modular model checking techniques betray their hardware roots: they assume that modules compose in parallel. In contrast, collaboration-based software designs, which have proven very successful in several domains, are sequential in the simplest case. Most interesting collaboration-based designs are really quasi-sequential compositions of parallel compositions. These designs demand and inspire new verification techniques. This paper presents algorithms that exploit the software's modular decomposition to verify collaboration-based designs. Our technique can verify most properties locally in the collaborations; we also characterize when a global state space construction is unavoidable. We have validated our proposal by testing it on several designs.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Modules and interfaces; D.2.4 [**Software/Program Verification**]: Model checking; D.2.11 [**Software Architectures**]: Languages; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification Techniques

General Terms

Design,Verification

Keywords

Model checking, compositional reasoning, computer-aided verification, software architecture, collaboration-based design, aspect-oriented programming

1. INTRODUCTION

Software designs can be decomposed into *actors*, *roles* and *features*: actors play roles to collaboratively implement features. For example, the actors might be databases, Web interfaces and control logic; features include on-line shopping and inventory management. Software designs must provide a coherent organization for the code implementing actors and features. Traditional software organizations arrange programs around actors: each module reflects an actor, and the collection of actor modules forms the complete design.

Recent research suggests that, in many domains, organizing designs around features rather than actors produces more reusable designs and implementations. In such a design, each module reflects a feature, and contains fragments of actors playing the roles relevant to implementing that feature. As with aspects [24], these designs often divide the implementation of each actor into several fragments that cross-cut the modular structure of the design. Modules in such designs are sometimes called *collaborations* [27], *hyperslices* [29], *refinements* [5], or *units* [14]. We will use the term *collaborations*, but our work applies equally well to these other forms of aspect-like, cross-cutting-based designs.

Collaborations have well-defined interfaces that permit their composition to build larger designs; they tend, at least in principle, to obey the characteristics of components [17, 21, 36], such as separate compilation, multiple instantiability and external linkage. By specifying each collaboration independently, designers can flexibly decide which features to include and exclude in a particular composite design. These designs have been particularly successful in software product lines, where different compositions of a collection of features can produce several variations on a theme. A brief sampling of successful designs in this vein includes a military command-and-control scenario simulator [4], a programming environment [13], protocol layers and database modules [5, 6, 38], and verification tools [16, 35].

The success of collaboration-based designs at *implementing* software product lines suggests a tantalizing prospect: perhaps they can also assist in *validating* such designs. Each collaboration's interface would indicate properties that hold of that collaboration. At composition time, the designer would validate these interface properties against other collaborations, thus ensuring they hold over the entire design. This scenario represents a useful and important form of modular verification.

While verifying designs based on their modular structure is an old idea, collaboration-based designs do not appear to fit existing modular verification frameworks. Most modular verification literature concentrates on parallel composition of modules, as befits its hardware origins. There is a small corpus on sequential composition, which represents a very simple case of collaboration-based design. Realistic collaboration-based models of software, however, quasi-sequentially compose collaborations of parallel execution. This subtlety requires a new verification methodology. This structure, moreover, would simplify the thorny questions that arise in traditional modular verification of how to perform property decomposition.

In this paper, we propose a verification methodology for collaboration-based software designs. We use model checking as our underlying verification technique. Our methodology checks properties of individual collaborations before composition; it also generates constraints to verify against other collaborations during composition. In most cases, especially the ones we have encountered in practice, we can establish these properties without ever constructing a global state space. This reuses the verification effort on individual collaborations, and avoids the horrors of state explosion.

Our efforts to verify a fire simulation support software package called FSATS [4] have inspired and driven our results. The FSATS implementation is a real and substantial software product, in which the collaboration-based architecture is fundamental to the design's development and maintenance. As FSATS is too complex to serve as a running example in this paper, we illustrate our development on two simple examples that distill the problems that we have encountered to date in our work on FSATS.

The rest of this paper is organized as follows. Section 2 discusses prior work on modular verification and its relationship to our work. Section 3 presents our methodology. Section 4 presents conclusions and discusses avenues for future work.

2. BACKGROUND AND RELATED WORK

Model checking is a technique for proving logical properties of designs [9]. Its successful application to hardware makes its use on software designs an attractive proposition. In a canonical model checker, a design is represented as a (finite) state machine, while properties are usually expressed in variants of temporal logic. Model checkers handle designs consisting of several machines running in parallel by automatically computing the cross-product of the machines, then applying their algorithms to the resulting single machine; we exploit this feature in section 3. For an extensive survey of model checking, we refer the reader to the book by Clarke, Grumberg and Peled [9]. In the rest of this paper, we assume a basic familiarity with model checking.

Model checking algorithms vary with the logic of properties. Our work extracts properties of collaborations by examining the labels on interface states. This assumes the model checker uses state labeling, which is the technique employed for branching-time temporal logics such as CTL. To simplify the development, we present our algorithms assuming an explicit representation of the state space of a design. In practice, many model checkers represent state symbolically rather than explicitly [28]. Our algorithms are insensitive to this difference; indeed, we performed the verification tasks in this paper on a model checker employing symbolic representations [37].

Several researchers have described techniques for modular verification of designs [15, 20, 25, 30]. These techniques are based on a hardware-oriented notion of modularity, in which modules are composed in *parallel*. For instance, one module might be a CPU, while another module represents a floating-point co-processor. The research then shows how to ensure the preservation of individual properties about the CPU or floating-point processor; using these techniques to prove properties involving both devices requires substantial experience, and is not always possible. These results do not apply to most software designs, where control flows sequentially between modules.

Some preliminary research [2, 10, 26] has begun to consider modular verification with sequential, rather than parallel, control flow. The original work [26] handles designs with only one state machine; it also lacks a design framework, such as collaboration-based design, that drives the decomposition of the design. Subsequent work uses hierarchical state machines [2] and StateCharts [10] to provide this decomposition, but the resulting designs are still monolithic. In contrast, we analyze designs with two key distinguishing features:

- We have to verify individual collaborations without knowing about all the other collaborations that may exist. In other words, we need to verify open, rather than closed, systems.
- The designs include *multiple state machines per collaboration*, which greatly complicates the verification problem.

The work by these other authors does not even admit these design possibilities. Alur and Yannakakis cite the problem of sequential verification over multiple state machines as open for future work [2]. Furthermore, they do not discuss how to handle designs that involve quasi-sequential composition of parallel compositions, such as exist in FSATS. Alur *et al.* discuss analysis techniques for sequential refinements within modules that are composed in parallel (this work uses the term "behavioral hierarchy" for refinements within modules and "architectural hierarchy" for parallel compositions of modules) [1]. The critical difference between their work and ours is that theirs does not support *coordination* between sequential refinements across modules. Our work, in contrast, considers verification for collaborations that gather related sequential refinements into modules. Encapsulating related refinements in collaborations allows us to verify properties of entire features in isolation from other features, even when those features cross-cut several actors. Without a collaboration-based architecture, isolating this information from across parallel modules is difficult if not impossible.

Inverardi, Wolf, and Yankelevich extract graph-based representations of software components for purposes of checking for deadlock-freedom [23]. Our work relates to theirs in that we also represent components with graph-based abstractions and traverse those graphs to check properties. Our work differs from theirs in two key ways: first, we are establishing an overall methodology for handling collaboration-based design, which adds some technical challenges to the verification problem (described in the remainder of this paper), and second, we support a larger class of properties (namely those representable in CTL).

Much software architecture research has focused on support for product-lines and separation of concerns. Some of this work uses *layered architectures* [32], whose high-level structure resembles collaboration-based architectures; layered architectures, however, generally include an assumption that each layer refines a more abstract layer already in the system. Our work does not require any abstraction relationship between collaborations. Rajlich and Silva studied software reuse and evolution of *orthogonal architectures*, which organize code fragments into *layers* of code at the same level of abstraction and *threads* of code for common actors [31]. Their work did not consider formal analyses of designs under orthogonal architectures. Griswold and Notkin explore performance issues arising from layered architectures [19]. While their layers resemble collaborations, their "actors" were data abstractions (or *views*) on shared data in a design with loose coupling between the pieces of a layer. Our actors, in contrast, are control-intensive and are tightly coupled within each collaboration. In general, the loose coupling in collaboration-based architectures occurs between, rather than within, the "layers".

3. VERIFYING COLLABORATION-BASED SOFTWARE DESIGNS

3.1 A Model of Collaboration-Based Design

We view a design as a set of classes, roughly one per actor in the design. A collaboration consists of a set of class extensions (*mixins* [7, 18, 33, 34, 39]) for the actor classes. The set of mixins in a collaboration relate to a common task, or *feature*, in the overall design. This definition permits actor classes 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 that each mixin extends an existing (base) state machine by adding nodes, edges, and/or paths between states in the base machine. State machine models of software arise from one of two sources: either the software is written in terms of state machines, as is true for many embedded software applications, or abstraction techniques derive state machines from the source code [11, 12, 23]. As FSATS is of the former flavor, we assume that approach in this paper. Our work could adapt to the latter if the abstractions produce machines for which we could define meaningful interfaces between collaborations; accordingly, we regard the work on state machine abstractions as orthogonal to this paper.

Each base or composed design specifies interfaces, in terms of states, at which clients may attach extensions. We define interfaces formally below. In our experience, new features generally attach to the base design at common or predictable points; the set of interfaces is therefore small. This is important, as the interface states will indicate information that we must gather about a design in order to perform compositional verification of collaborations; a large number of interfaces might require too much overhead in our methodology.

Figures 2 and 5 show examples of base designs, collaborations, extensions, and interfaces; Sections 3.3 and 3.4 explain the examples in detail. The following formal definition makes our model of collaboration-based designs precise. The definitions match the intuition in the figures, so a casual reader may wish to skip the formal definition.

Definition 1. A *state machine* is a tuple $\langle S, \Sigma, \Delta, s_0, R, L\rangle$, where S is a set of states, Σ is the input alphabet, Δ is the output alphabet, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \times S$ is the transition relation (where $PL(\Sigma)$ denotes the set of propositional logic expressions over Σ), and $L : S \rightarrow 2^{\Delta}$ indicates which output symbols are true in each state.

Definition 2. A *base system* is a tuple $\langle M_1, \ldots, M_k\rangle$ of state machines and a set of *interfaces*. We denote the elements of machine M_i as $\langle S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s_{0_{Mi}}, R_{Mi}, L_{Mi}\rangle$. An interface contains a sequence of pairs of states

$$\langle\langle exit_1, reentry_1\rangle, \ldots, \langle exit_k, reentry_k\rangle\rangle.$$

Each $exit_i$ and $reentry_i$ is a state in machine M_i. State $exit_i$ is a state from which control can enter an extension machine, and $reentry_i$ is a state from which control returns to the base system. Interfaces also contain a set of properties and other information which are derived from the base system during verification; we describe these properties in detail in later sections.

Definition 3. An *extension* is a tuple $\langle E_1, \ldots, E_n\rangle$ of state machines. Each E_i must induce a connected graph, must have a single initial state with in-degree zero, and must have a single state with out-degree zero. For each E_i, we refer to the initial state as in_i and the state with out-degree zero as out_i. States in_i and out_i serve as placeholders for the states to which the collaboration will connect when composed with a base system. Neither of these states is in the domain of the labeling function L_i.

Given a base system B, one of its interfaces I, and an extension E, we can form a new system by connecting the machines in E to those in B through the states in I, as shown in Figure 1. For purposes of this paper, we assume that B and E contain the same number of state machines. This restriction is easily relaxed; the relaxed form allows actors to not participate in each new feature, or to allow new actors as required by new features. We also assume that the states in the constituent machines of base systems and extensions are distinct.

Definition 4. Composing base system $B = \langle M_1, \ldots, M_k\rangle$ and extension collaboration $E = \langle E_1, \ldots, E_k\rangle$ via an interface I=$\langle\langle exit_1, reentry_1\rangle, \ldots, \langle exit_k, reentry_k\rangle\rangle$ yields a tuple $\langle C_1, \ldots, C_k\rangle$ of state machines. Each state machine $C_i = \langle S_{Ci}, \Sigma_{Ci}, \Delta_{Ci}, s_{0_{Ci}}, R_{Ci}, L_{Ci}\rangle$ is defined from $M_i =$

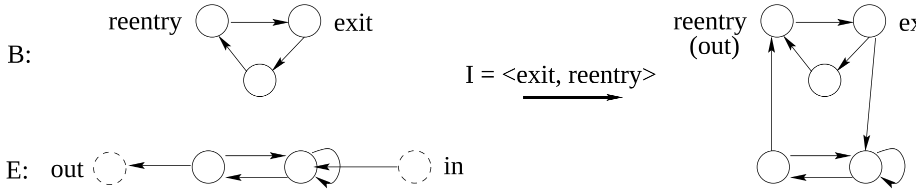

Figure 1: Composition of a base system B with an extension E via an interface.

$\langle S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s0_{M_i}, R_{Mi}, L_{Mi}\rangle$ and its corresponding extension $E_i = \langle S_{Ei}, \Sigma_{Ei}, \Delta_{Ei}, s0_{E_i}, R_{Ei}, L_{Ei}\rangle$ as follows: $S_{Ci} = S_{Mi} \cup S_{Ei} - \{in_i, out_i\}$; $s0_{C_i} = s0_{M_i}$; R_{Ci} is formed by replacing all references to in_i and out_i in R_{Ei} with $exit_i$ and $reentry_i$, respectively, and unioning it with R_{Mi}. All other components are the union of the corresponding pieces from M_i and E_i. We will refer to the cross-product of $C_1, \ldots, C_k$ as the *global composed state machine.*

Definition 4 allows composed designs to serve as subsequent base systems by creating additional interfaces as necessary. This supports the notion of compound components that is fundamental in most definitions of component-based systems.

3.2 Verification Methodology

Our methodology is designed to support compositional verification of collaboration-based designs. Specifically, our methodology supports the following activities:

1. Proving a CTL property of an individual collaboration or composition of collaborations. This is easily done in the base system with existing techniques, but becomes more complicated in extension collaborations.
2. Deriving a set of constraints on the exit and reentry states of a collaboration that are sufficient to preserve a particular property after composition (the *preservation constraints*).
3. Proving that a collaboration satisfies the preservation constraints of another collaboration (or existing system). This activity is only meaningful if the preservation constraints were generated for the exit and reentry states to which the new collaboration will attach. We establish preservation by analyzing only the extension, not the composition of the extension and the existing system.

These activities correspond to a kind of modular verification, where the collaborations are modules. As in standard approaches to modular verification, we are interested in proving properties of modules in isolation from the rest of the system and in preserving those properties upon composition with other modules.

We illustrate our methodology using two examples: a simple sportswatch and a communication protocol. The sportswatch design consists of a single actor; each collaboration therefore contains and extends only one state machine. This example motivates our interfaces and high-level approach to sequential collaboration composition. The communication protocol captures key characteristics of FSATS and shows how our methodology extends to designs with multiple state machines in each collaboration. Section 3.5 discusses some pragmatic issues that arise when performing these verification runs with an existing model checker.

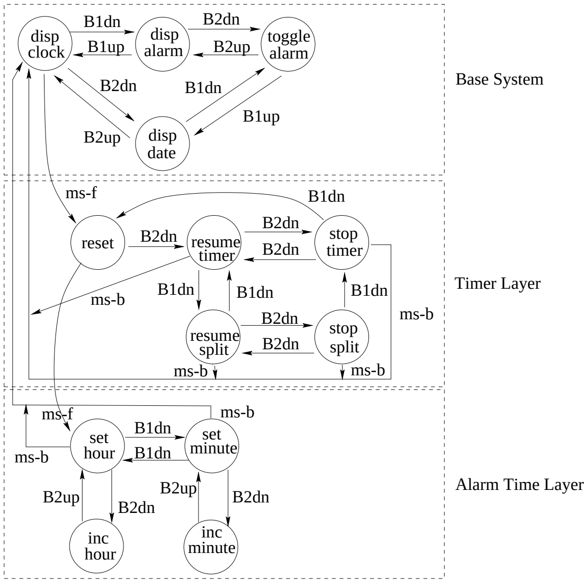

Figure 2: A collaborative design for a sportswatch.

3.3 Single-Machine Designs

Figure 2 shows a collaboration-based design of a sportswatch with timer and alarm features. The base system contains four display nodes: clock display, alarm time display, date display, and an alarm status display that supports toggling the alarm status. The first extension adds a timer which the user can reset, resume, and stop. The timer collaboration also supports a split timer for capturing time instantaneously. The second extension supports setting the alarm time; we omit collaborations for setting the clock time due to space constraints. Although both extensions add core functions, rather than optional features, we implement them as collaborations to allow a designer to include any of several possible implementations of these features in a final watch (as in a product-line architecture). The watch is controlled through two buttons (*B1* and *B2*) which can be either up or down and a mode switch that can be in the forward (*ms-f*) or back (*ms-b*) positions.

The base system should satisfy the property that one can always get to the display-clock state (AG EF *display_clock* in CTL). This property is easy to verify using a model checker. The base collaboration publishes one interface: $\langle \mathit{dispclock}, \mathit{dispclock} \rangle$, meaning that all extensions will start from and return to the *dispclock* state. Once we extend the base system with the timer, we must prove that adding the timer will not cause the display-clock property—which has already been proven of the base collaboration—to fail. We could compose the base system and timer collaborations and re-verify the property on the composed system. This approach, however, wastes the work that we have already done proving the property of the base collaboration; worse still, on a larger example, the composed design could be too large to model check effectively. We therefore want to verify that the timer collaboration will preserve the property already proven of the base system without using the entire base system.

The classic CTL model checking algorithm [8] checks a property by marking each state with the subformulas of the property that are true in that state. After marking is complete, the formula is true of the design if its initial state is marked with the full property formula. If we can prove that an extension does not alter the markings of the base system states for a given property, then that property will hold in the composition of the base system with the extension as well. It suffices to show that the markings of the exit states in the base system interfaces are not altered, as all states which reach collaboration states do so through the exit state.

Given the base system interface ($\langle \mathit{dispclock}, \mathit{dispclock} \rangle$ in this case) and a property to preserve ($\mathsf{AG}\ \mathsf{EF}\ \mathit{display\_clock}$), we use a model checker to extract the set of subformulas of the property that mark each state in the interface; these markings can be stored with the interface, and need not be re-computed on each extension. The following three formulas mark *dispclock*:

- $\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{dispclock})$
- *dispclock* (this implies the previous formula)
- $!(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ !(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{dispclock}))))$ (equivalent to $\mathsf{AG}(\mathsf{EF}\mathit{dispclock})$).

We must prove that the extension will preserve the markings on the exit state from the base system. The CTL model checking algorithm marks states based on the markings of its successor states. As some extension states have transitions to the reentry state (in the base system), we need the reentry state's markings to compute the markings on the extension states. Our verification algorithm consists of assuming that the *out* state of the extension has the same markings as the reentry state, deriving the markings on the *in* state, and checking that those markings are the same as on the original exit state. We derive the markings on the *in* state by checking a property of the form $\mathsf{AG}(\mathit{in} \rightarrow\ \phi)$ for each subformula ϕ of the property to be preserved. Figure 3 shows the sportswatch timer collaboration with the marking assumptions on *out*. Model checking confirms that *in* retains the original markings of *dispclock*, so the property will be preserved upon composition.

In addition to the display-clock property, we can also verify that the timer collaboration (without the base collaboration attached) satisfies the property "once started, the timer can always be stopped" ($\mathsf{AG}(\mathit{resumetimer} \rightarrow \mathsf{EF}\ \mathit{stoptimer})$). We view the timer collaboration as the base system and the base as the extension to verify that the base collaboration would preserve this property upon composition.

We also construct a composed system from the base and timer collaborations, with interface $\langle \mathit{dispclock}, \mathit{reset} \rangle$. The interface states change after composition because the watch requires switching between modes to be deterministic; satisfying this constraint requires new collaborations to be entered from the timer collaboration, rather than the original base system. For both states in the interface, we record the markings necessary to satisfy the two properties already proven of the design. These markings arise from both verifying the properties of each collaboration and from verifying the preservation of the other collaboration's properties. For *dispclock*, the new set of interface markings is:

- $!(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ !(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{dispclock}))))$;
- $\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{dispclock})$;
- *dispclock*;
- $!(\mathsf{E}(\mathit{TRUE}\,\mathsf{U}\ !((\mathit{resumetimer} \rightarrow \mathsf{E}(\mathit{TRUE}\,\mathsf{U}\ \mathit{stoptimer})))))$;
- $(\mathit{resumetimer} \rightarrow \mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{stoptimer}))$;
- $\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{stoptimer})$

Using these markings, we verify that adding the alarm collaboration preserves the existing properties (displaying the clock and stopping the timer).

Summary of Algorithm on Single State Machines

In summary, the verification algorithm for the single state machine case is as follows:

1. Write the model for the extension, including the placeholder states *in* and *out*.
2. Assign the subformulas that marked the actual reentry state in the base system as labels of the placeholder reentry state (*out*).
3. Model check all of the subformulas of the original property in the placeholder reentry state (*in*). If *in* has exactly the same labels (relative to the property) as it did before the extension, the property will hold in the composed system.

This algorithm was independently derived by Laster and Grumberg for reasoning about sequential decomposition of finite state machines [26]. Its correctness depends in part on all reachable states in the composed design lying in either the base system or the extension (an obvious point in the single-machine case, but one which becomes interesting in the multiple-machine case). For checking preservation of purely existential properties, this algorithm is often unnecessary because sequential composition trivially preserves such properties when the out state is always reachable from the *in* state (a simple observation, but one which Laster and Grumberg failed to note).

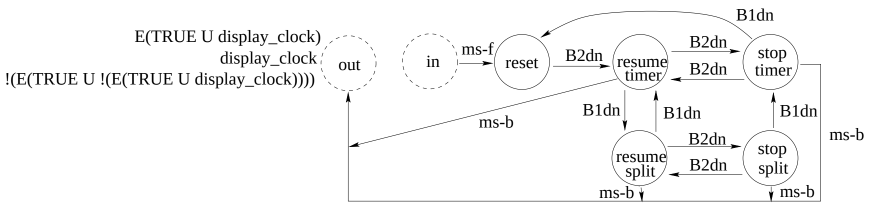

Figure 3: The timer extension with marking assumptions on the *out* state.

For our experiments, we simulated this algorithm using the VIS model checker. VIS does not support this algorithm directly, as there is no way to seed *out* with the assumed marking. Instead, we were forced to include a transition from *out* to the entire base system model; we did not include transitions from the base system to *in*. We verified that the markings on the actual reentry state (*dispclock*) did not change under this operation. As *in* was not attached to the base system, this approach is sufficient to argue that the verification would have gone through with the seeded markings (and no base system) had VIS supported that operation. Section 3.5 discusses our experiences using a conventional model checker for our sort of modular verification in more detail.

3.4 Multiple-Machine Designs

The algorithm in Section 3.3, as well as prior research into verification under sequential composition, does not apply to FSATS because FSATS has multiple state machines in each collaboration. In practice, almost all interesting collaboration-based designs, by their very nature, will employ multiple state machines (one for each actor). When each collaboration contains a single state machine, extending a design with a collaboration corresponds to sequential composition of state machines. When collaborations contain multiple state machines, extending a design with a collaboration corresponds to a hybrid of sequential and parallel composition: the machines within a collaboration are composed in parallel (because they run together to implement a particular feature), but the collaborations themselves are composed in a quasi-sequential manner (quasi because the machines do not synchronize exactly when entering an extension). The actual composition is not strictly sequential: this detail is at the crux of the verification problem for designs like FSATS, and is the focus of this section.

Constructing a design by sequential composition is appealing because, as Section 3.3 shows, it supports independent verification of collaborations. Figure 4 (left) shows a design constructed in this fashion. The construction provided in the formal model (the global composed state machine, Definition 4), however, is different. As Figure 4 (right) illustrates, the construction first extends each base machine with its corresponding mixin, then composes the resulting machines in parallel. Clearly, we would prefer to compose designs according to the first construction because it supports collaboration-based verification. In order to do this, however, the first construction must produce the same global composed state machine (upto reachability of states) as the second! This relationship captures the crucial challenge in collaboration-based verification of designs with multiple state machines per collaboration. We must construct the parallel compositions representing each collaboration in such a way that composing them sequentially yields the global state machine arising from Definition 4.

This section motivates our algorithm for constructing parallel compositions within collaborations. Our algorithm is designed to create parallel compositions that can in turn be composed sequentially with other collaborations. We describe the algorithm by illustrating its behavior on a small example. We also evaluate this algorithm's ability to verify properties of collaborations in isolation. While many collaborations (including the FSATS collaborations) can be verified in isolation under this construction, our motivating example illustrates a case where independent verification may fail. A property for which verification may fail must be verified in the composed design, rather than compositionally through the collaborations. We provide a characterization of these cases and a model-checking-based algorithm to determine whether properties can be verified compositionally. Section 3.4.1 presents our new example, which captures the salient characteristics of FSATS without necessitating as much explanation of the domain.

3.4.1 The Clayton Tunnel Protocol

We consider a collaboration-based design of a communications protocol between operators at either end of a train tunnel (see Figure 5) [22]. Our design is derived from an actual communication protocol that was in use (and contributed to an accident!) in England in 1861. The two state machines model the human operators on either end of long train tunnel covering a one-way track. Unable to see one another, the operators communicate messages about the status of the tunnel. In the base collaboration, the operators communicate when trains are entering and exiting the tunnel. The inbound operator sends a *train-in* message to the outbound operator when a train enters the tunnel. The outbound operator sends a *tunnel-clear* message to the inbound operator when a train exits the tunnel. The base collaboration consists of the protocol for exchanging these two messages.

The full protocol was designed to prevent two trains from ever being in the tunnel simultaneously (we omit the specific details from the model is this paper because they are irrelevant for our purposes). The accident that occurred arose because a second train entered the tunnel before the first one had left; although the inbound operator suspected the problem, the communication protocol was too weak to convey the situation to the outbound operator. One solution is to add messages to the protocol that convey this information accurately. The extension adds a *two-in* message from the inbound to the outbound operator; it also adds states to

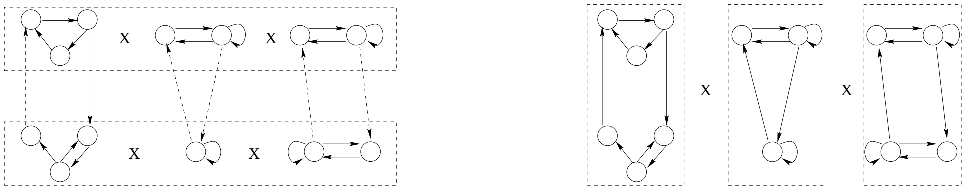

Figure 4: Two approaches to constructing composed systems.

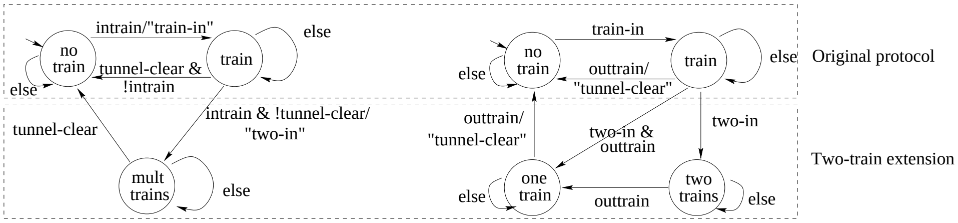

Figure 5: A collaboration-based design for a track-operator communication protocol.

both operator machines so that the outbound operator does not send the *train-clear* message until both trains have left the tunnel.

Verifying this protocol requires a model of the trains that can enter and exit the tunnel. A model of the events that drive a protocol, but are not part of its definition, is called an *environment model*. The environment model for the tunnel protocol must generate reasonable train data; for example, no train should ever leave the tunnel before it enters the tunnel. For simplicity, we use an environment model containing two trains. Their only constraints are that the first train enters the tunnel before the second, and that both trains enter the tunnel before they exit the tunnel. This model is reasonable because the original protocol was such that at most two trains could be in the tunnel at once if the train drivers obeyed the rules of using the tunnel. We implement environment models as state machines. For the tunnel protocol, the environment model generates signals *intrain* and *outtrain* to indicate trains entering and leaving the tunnel.

Depending upon when trains enter and leave the tunnel, the operators may be inconsistent on their views as to whether there is a train in the tunnel. Given the base collaboration, we would like to prove that the inbound operator never livelocks thinking that there is a train in the tunnel ($\mathsf{AG}(\mathsf{EF}(\mathit{inbb.state}{=}\mathit{notrain})))$; this property requires all trains in the tunnel to eventually exit the tunnel, which we handle with a fairness constraint [9]. We can easily discharge this property of the base system; the challenge is to verify that the extension preserves it. For the extension, we wish to prove that once the inbound operator warns that there are two trains in the tunnel, it does not exit the extension until it receives a tunnel-clear message ($\mathsf{AG}\ ((\mathit{inbmsg}{=}\mathit{two\text{-}in}) \rightarrow \mathsf{A}(!(\mathit{instate}{=}\mathit{out})\ \mathsf{U}\ (\mathit{outbmsg}{=}\mathit{tunnel\text{-}clear}))))$.

3.4.2 *Creating the Extension Cross-Product*

The extension consists of the two state machines in the lower dashed box in Figure 5 (though with *in* and *out* states, as in Figure 3). In order to model check the extension, we need to compose the extension machines in parallel. We could form a naïve parallel composition of these two machines using a standard cross-product procedure [9]. This construction would assume that both machines start in their initial states (the *in* states) simultaneously. This assumption, however, is not necessarily valid. In the tunnel protocol, for example, the inbound operator may notice the second train before the outbound operator has registered that there is a train in the tunnel (this synchronization problem arises in FSATS). Our parallel composition therefore needs additional information about the synchronization of the *in* states in the extension in order to construct a valid composition; without this information, we may use the wrong initial states during the parallel composition within the extension.

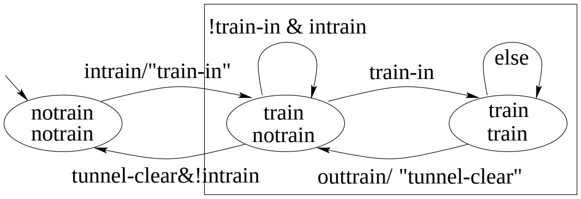

Figure 6: The cross-product state machine (reachable subset) for the tunnel base collaboration. The exit subgraph for an interface containing both *train* states as exit states is enclosed in the solid box. Both states in the exit subgraph have the potential to transition to an extension.

This synchronization observation reflects a general technical challenge with collaboration-based verification: the reachable global state space of an extended design may contain global states comprised of states from both the base system and the extension. Our techniques must guarantee that we visit all such states during model checking. As most collaboration-based designs roughly synchronize actors around each collaboration, these hybrid states arise in two controlled places: as the actors enter the extension and as the actors leave the extension to return to the base system. Our techniques must identify these hybrid states and use them both to properly generate the parallel composition of an extension and to check whether the synchronization is sufficient to avoid model checking the entire system. Section 3.4.4 discusses the latter issue.

We derive synchronization information on the exit states from the base system. Given a set of exit states that form an interface in the base system, we can compute the subgraph of the base system that involves only the exit states; we then use this subgraph (the *exit subgraph*, described formally in Definition 5) to drive transitions from the *in* states in the parallel composition. Figure 6 shows the exit subgraph for the tunnel protocol. While in practice the exit subgraph could be large, these graphs are small in FSATS (and presumably in similar designs as well) because the actors decide to enter a particular extension at roughly the same time based on a tight sequence of message-passing.

Definition 5. Given the cross-product B of the base system and an interface

$$I = \langle\langle exit_1, reentry_1\rangle, \ldots, \langle exit_k, reentry_k\rangle\rangle$$

to the base system, the *exit subgraph* is the subgraph of B over all cross-product states containing at least one $exit_i$ state.

The exit subgraph indicates which states from the base system could be involved in transitions to the extension. Intuitively, each state in the exit subgraph is a potential initial state for the cross-product of the extension collaboration. We must therefore drive the construction of the extension cross-product from all states in the base system that appear in some state of the exit subgraph. To do this correctly, we must prepend each individual machine in the extension with states from the exit subgraph.

Figure 7 illustrates the needed construction. The diagram on the left shows an exit subgraph and a potential state in the extension (note that state $\langle in_1, in_2\rangle$ is not reachable). In order to construct the potential extension state from its correct predecessor, we need to capture states b_1 and b_2 in the extension machines before forming the cross-product. The diagram on the right shows the extension machines E_1 and E_2 expanded with information obtained by decomposing the exit subgraph. The expanded machines are then used to build the extension cross-product.

The following steps formally describe how to construct the cross-product of the extension using the exit subgraph:

1. Construct the exit subgraph ES.
2. For each extension machine E_i:
   (a) Determine all states of base machine M_i that appear in ES.
   (b) Add these states to E_i (with the exception of $exit_i$, which is already in E_i as state in_i—we refer to $exit_i$ and in_i interchangeably throughout this description).
   (c) For each pair of states s_i and t_i added to E_i (including $exit_i$), add a transition from s_i to t_i iff such a transition exists in the base machine M_i. The guards on the transition should be identical to those on the transition from s_i to t_i in the base system machine.
   (d) Add a state $escape_i$ to E_i. For each state s_i from M_i added to E_i (including $exit_i$), add a transition from s_i to $escape_i$ enabled on all other transitions from M_i that leave s_i.
3. Treating each state in ES as a potential initial state, construct the cross-product of the expanded E_i machines using the standard cross-product construction. We will use this expanded cross-product for all model checking activities for the extension.

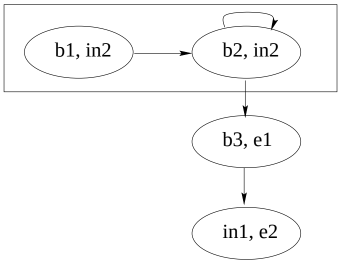

Figure 8: Using escape states to detect insufficient synchronization of interface states.

This construction yields a set of cross-product states and transitions involving states from the original extension machine E_i. The composed cross-product machine arising from Definition 4 also yields a set of cross-product states and transitions involving states from the original extension machine E_i. Our construction is correct if these two sets of states and transitions are identical. The correctness proof argues (a) that the states leading into the extension are the same under both constructions, and (b) that since these states (the initial states of the extension) are the same under both constructions and the transition labels on the individual machines are identical, the generated sets of states and transitions must also be identical. The proof requires an assumption that no cross-product state involving an $escape_i$ state is reachable under our construction. We restrict our methodology to cases where this condition holds in Section 3.4.4.

Ideally, this construction should yield all cross-product states in the composed design that arise from entering the extension. This situation might not hold in the general case, however, as illustrated in Figure 8. In the diagram, the second machine enters the extension before the first machine reaches its interface state. This creates a global state $\langle b_3, e_1\rangle$ which spans the base system and the extension, but without involving an interface (*exit*) state. This example motivates the use of the *escape* state in the extension cross-product construction. The construction would yield a transition from $\langle b_2, in_2\rangle$ to $\langle escape_1, e_1\rangle$ (where $escape_1$ captures state b_3). Reachable *escape* states capture cases such as this in which our methodology could not correctly apply.

3.4.3 Environment Models for Verifying Extensions

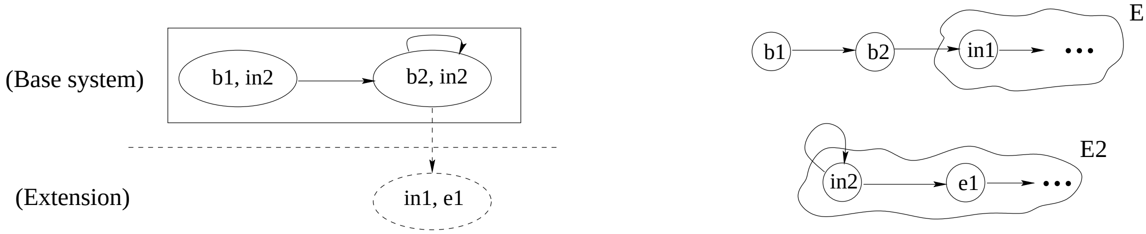

Figure 7: Decomposing an exit subgraph to drive extension cross-product construction.

Given the parallel composition of the extension machines constructed using the exit subgraph, we can attempt to verify the collaboration property using the original environment model to generate the trains. This effort fails. The inbound operator sends the *two-in* message as soon as the environment model sends the *first* train into the tunnel; this is wrong, however, because the inbound operator should only enter the multiple train state when the *second* train enters the tunnel before the first train exits. Aligning the initial states of the extension cross-product with the initial states of the environment model loses some history about the state of the environment model at the interface states to the extension. In the tunnel example, the environment model must have the first train in the tunnel and the second train approaching the tunnel at the *in* states of the extension; the normal environment model starts with both trains approaching the tunnel. We can synchronize the environment model with the extension by composing the environment model with the base system before computing the exit subgraph. The initial states of the exit subgraph now contain states of the environment model; those states should be used as the initial states of the environment when verifying properties of the collaboration. This construction indicates that the tunnel environment should start with the first train already in the tunnel.

Although generating restricted initial states of the environment model appears to be an overhead of formal verification, the problem of generating these models is similar to the problem of generating a testing harness for a collaboration-based design. Collaboration-based designs offer the hope of testing collaborations in isolation. That testing, however, requires knowledge about the environment that will drive the collaboration. Our approach merely formalizes the problem of obtaining a restricted testing harness for collaboration-based designs. In FSATS, the environment model problem arises because each extension corresponds to a new type of mission which is initiated only if the environment has generated a target of a particular type.

3.4.4 Verifying Properties Compositionally

We have identified two key issues in supporting verification of multiple-machine collaborations independently from their base systems: capturing global states that bridge collaborations and restricting environment models. Exit subgraphs and restricted environment models allow us to verify that an extension satisfies a given property relative to an interface to a base system. The methodology as presented is still incomplete, however, as we must characterize *when* the properties of the composition of this collaboration with a base system can be verified via sequential composition, rather than on the global composed state machine. Verifying multiple-machine collaborations under sequential composition is correct only if we are able to identify and capture all global states that contain sub-states from both the base and extension collaborations. Thus, we require sufficient conditions for capturing these states.

Section 3.4.2 described a construction for the exit subgraph, which coordinates actors as they enter an extension. Just as the actors do not enter an extension simultaneously, we cannot expect them to exit an extension collaboration simultaneously. The asynchronous exits could create reachable states in the composed system that are not contained in the extension. Worse still, these states could lead to global states that become reachable in the base system only after composition. Either case would break our proposed collaboration-based verification methodology.

Fortunately, the collaboration-based designs that we have studied, including FSATS, tend to have a characteristic that addresses this problem: the reentry states *eventually* synchronize after executing an extension. With this synchronization, the sequential composition of the base system and the extension could capture the full global state space, as required for collaboration-based verification. If this synchronization does not occur, then our methodology is insufficient; in such cases, we will have to check the properties in the full composed system.[1]

The following constraints indicate when we can prove that a collaboration preserves a property under sequential composition. The first two address issues related to entering an extension, while the latter two address issues related to reentering the base system.

1. Every reachable state in the extension cross-product contains some substate (*in* qualifies) from the original extension machine.

2. No reachable state in the extension cross-product contains an $escape_i$ substate.

3. The *reentry* states eventually synchronize. That is, the state $\langle out_1, \ldots, out_k \rangle$ is reachable and terminal in the extension cross-product.

4. State $\langle reentry_1, \ldots, reentry_k \rangle$ is reachable in the base system cross-product.

[1] When we need to verify in the full composed system, we can apply existing techniques for parallel composition. As these techniques can be very difficult to use in practice, applying them effectively remains an open problem.

These constraints restrict reentry to the base system more than the exit to the extension. Intuitively, this means that actors must synchronize more tightly when leaving a collaboration than when entering it. This seems necessary to avoid generating additional reachable states in the global base collaboration. It also enables us to reuse our existing verification methodology in which we use properties of states in the base system to check properties in the extension; requiring state $\langle reentry_1, \ldots, reentry_k \rangle$ to be reachable provides a concrete reentry state from which to seed the verification of the extension. We defer the proof that the given conditions are sufficient to prove a correspondence between the two constructions of the global composed state space. Intuitively, the proof consists of an argument that, under the above constraints, all reachable states under the first construction are reachable states in either the extension or the base system under the second construction. The interesting cases of this proof involve global states with some components in the base system and some in the extension. The conditions listed above restrict all such states to lie in the extension including the exit subgraph.

3.5 Implementation

We have conducted all the model checking tasks described in this paper using the VIS model checker [37]. We modified VIS slightly to display all sub-formulas of properties generated during the marking phases; we used these sub-formulas for verifying the preservation of properties in other collaborations. For the paper's examples, the time and space usage are negligible, so we do not report them.

Section 3.3 describes how we simulated the modular verification scenario while in fact attaching extensions to, potentially, the entire base system. This approach was necessary because existing model checkers do not appear to be designed for extension to verifying open systems. For instance, they do not provide a way to query and assert properties on specific states. Expressing our extension collaborations in Verilog (VIS's input language) required manual insertion of additional design variables because we could not easily unify states in the underlying symbolic transition system. Furthermore, the exit and reentry subgraphs were hard to connect to the extension collaboration in VIS's symbolic framework, so we constructed them manually. Computing the core subgraphs is straightforward (by adding routines to the VIS source code); adding the escape state is difficult because it requires us to essentially reverse-engineer the symbolic state encoding to find an unused boolean representation for the escape state. A front-end for supporting collaboration-based design languages could work around the limitations of Verilog, but the limitations of the symbolic framework are harder to surmount.

4. CONCLUSION AND FUTURE WORK

This paper has described how intricacies of collaboration-based software designs inhibit straightforward attempts at modular verification. We demonstrate that the standard decompositions of designs into parallel or sequential modules are insufficient for collaboration-based software design: verification under parallel composition is extremely difficult in practice and verification under sequential composition is too simplistic. In practice, many software designs tend to be quasi-sequential compositions of parallel compositions. We explain how certain constraints can make modular verification tractable by reducing such designs to purely sequential compositions of parallel compositions; these constraints are reasonable because many existing software designs appear to satisfy them.

In the big picture, our verification model attempts to parallel the software development model by respecting independent development followed by externally-specified composition. By analogy to separate compilation, we try to minimize the work expended in verifying compositions relative to the work done verifying the individual collaborations. Our technique can potentially apply to a variety of system designs including certain uses of components and aspects. It is especially applicable to collaboration-based product line designs.

We have concentrated solely on model checking because we want to understand the strengths and limitations of algorithmic verification on collaboration-based designs. Our experience suggests that extant model checkers have not been designed to be extended for such tasks. We therefore intend to develop custom model checkers for this effort. We believe this will be necessary to complete the verification of the entire FSATS suite. A related question is how to extend our approach to handle LTL formulas; for technical reasons, we have only considered CTL properties.

Within the realm of algorithmic verification, model checking may be overkill for verifying certain collaboration properties. For instance, simple properties that ensure a design always reaches a consistent state may not need extensive verification in an extension: simply showing reachability between the extension's *in* and *out* states often suffices (this relates to checking the requirement in our formal model that extensions yield connected graphs). These properties arise both in the examples presented in this paper and in FSATS. Therefore, there is clearly potential for applying more light-weight verification tools, such as reachability engines and type systems. We expect further work with a richer set of designs to help us identify when the full power of our current methodology is required.

Collaboration-based designs can benefit from a broader scope of verification techniques. Early work on analyzing dependencies between collaborations [3] must be formalized and incorporated into any validation framework. Addressing this problem may involve creating special architectural description languages for collaboration-based designs that capture these dependencies and simplify construction of the exit and reentry subgraphs. Finally, we have encountered collaboration-based designs involving complex data invariants that will likely be more amenable to theorem proving.

5. ACKNOWLEDGEMENTS

We thank Don Batory for several enlightening discussions on collaboration-based designs, Jia Liu for sharing his design for the sportswatch, Harry Li for useful comments on an earlier draft, the Automated Software Engineering Group at NASA Ames for an engaging exchange, and the reviewers for their helpful suggestions.

6. REFERENCES

[1] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchic reactive machines. In *International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 2000.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Symposium on the Foundations of Software Engineering*, pages 175–188, 1998.

[3] D. Batory and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, Feb. 1997.

[4] D. Batory, 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.

[5] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, Oct. 1992.

[6] E. Biagioni, R. Harper, P. Lee, and B. G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Symposium on Lisp and Functional Programming*, 1994.

[7] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, Mar. 1992.

[8] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[10] E. M. Clarke and W. Heinle. Modular translation of Statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science, August 2000.

[11] J. C. Corbett, 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.

[12] M. B. Dwyer 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.

[13] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001. To appear.

[14] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[15] B. Finkbeiner, Z. Manna, and H. Sipma. Deductive verification of modular systems. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 239–275. Springer-Verlag, 1998.

[16] K. Fisler, 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.

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

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[19] W. G. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructing tool. *IEEE Transactions on Software Engineering*, 21(4):275–287, April 1995.

[20] O. Grumberg and D. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

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

[22] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[23] P. Inverardi, A. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, July 2000.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[25] O. Kupferman and M. Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

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

[27] K. Lieberherr, D. Lorenz and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, March 1999.

[28] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[29] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM, 1999.

[30] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[31] V. Rajlich and J. H. Silva. Evolution and reuse of orthogonal architecture. *IEEE Transactions on Software Engineering*, 22(2):153–157, February 1996.

[32] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, 1996.

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

[34] G. L. Steele, Jr., editor. *Common Lisp: the Language*. Digital Press, Bedford, MA, second edition, 1990.

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

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

[37] The VIS Group. VIS: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, July 1996.

[38] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical Report 97-1638, Department of Computer Science, Cornell University, July 1997.

[39] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 1996.