10-29-2002

# Modular Verification of Feature-Oriented Software Models

Kathi Fisler
*Worcester Polytechnic Institute*, kfisler@cs.wpi.edu

Shriram Krishnamurthi
*Brown University*, sk@cs.brown.edu

# Modular Verification of Feature-Oriented Software Models[*]

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

October 29, 2002

**Abstract**

Most existing modular model checking techniques betray their hardware roots: they assume that modules compose in parallel. In contrast, feature-oriented software designs, which have proven very successful in several domains, are sequential in the simplest case. Most interesting feature-oriented 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 feature-oriented designs. Our technique can verify most properties locally in the features; 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;

**Keywords:** Model checking, compositional reasoning, computer-aided verification, software architecture, feature-oriented design, aspect-oriented programming

## 1 Introduction

Programs 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

[*]An abbreviated version of this paper appeared at the joint ESEC/FSE Conference on Foundations of Software Engineering, Vienna, Austria, September 2001. The work was partially supported by NSF grants ESI-0010064, ITR-0218973, and CCR-0132659.

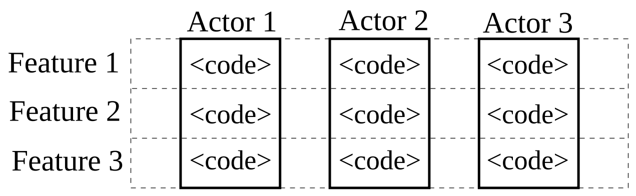


Figure 1: Actor-oriented versus feature-oriented decompositions.

of actors playing the roles relevant to implementing that feature. Figure 1 contrasts these two approaches. As with aspects [27], feature-oriented 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* [31], *hyper-slices* [35], *refinements* [5], or *units* [16]. We will use the term *features*, but our work should apply to other similar notions of cross-cutting.

Feature modules have well-defined interfaces that permit their composition to build larger designs; they tend, at least in principle, to obey the characteristics of components [19, 23, 42], such as separate compilation, multiple instantiability and external linkage. By specifying each feature 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 [12], where different compositions of a collection of features can produce 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 [15], protocol layers and database modules [5, 6, 45], and verification tools [18, 41].

The success of feature-oriented designs at *implementing* software product lines suggests a tantalizing prospect: perhaps they can also assist in *validating* such designs. Each feature module's interface would indicate properties that hold of that feature. At composition time, the designer would validate these interface properties against other features, thus ensuring they hold over the entire design. This scenario represents a useful and important form of modular verification. Furthermore, it is a potentially easier form of modular verification to use in practice than parallel composition. The latter requires complex decompositions of *properties* to align properties with modular boundaries [24]; with feature-oriented modules, properties and modules naturally align because both tend to originate from the same source, namely the program's requirements.

While verifying designs based on their modular structure is an old idea, feature-oriented 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 feature-oriented design. Realistic feature-oriented models of software, however, quasi-sequentially compose features encapsulating parallel execution. This subtlety requires a new verification methodology.

In this paper, we propose a verification methodology for feature-oriented software designs. We use model checking as the underlying verification technique. Our methodology checks properties of individual features before composition; it also generates constraints to verify against other features 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 features, and avoids the horrors of state explosion.

While these results are promising, the idea of widespread feature verification faces a difficult challenge. Modular verification depends on the existence of useful descriptions of interfaces. Experience suggests that developers are generally loath to ascribe complex interfaces to their modules; indeed, they seem to rarely express more formal properties than those captured by most programming language type systems. Verification of behavior, however, requires fairly sophisticated properties at module interfaces. Ascribing these properties to modules is not only onerous an an activity, it is also difficult to perform correctly. Performing this automatically is difficult due to the large variety of properties a module may satisfy, and the inherent decidability problems of determining these properties. What hope, then, is there for modular verification?

Fortunately, our technique automates the problem of property ascription at interfaces. Of the many properties that may be true (and false) of a module, only a few are of interest for a particular verification task. The key, then, is to not attempt to generate the “most general” interface, but rather to customize the interface to the verification task. In short, we advocate a strategy of *property-driven interface generation*. A new set of properties will require a new round of interface generation, but this is a small price to pay for automating a task developers are unlikely to perform. As a result, this approach represents a kind of “sweet spot” between manual attribution of simple interface properties (such as types) and the intractable peaks of inferring general interfaces.

We have successfully subjected property-driven interface generation to evaluation on a military mission simulator called FSATS [4]. Our efforts to verify FSATS have inspired and driven our results. The FSATSimplementation is a real and substantial software product, in which the feature-oriented architecture is fundamental to the design’s development and maintenance. As FSATSis 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 [10]. 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 [10]. In the rest of this paper, we assume a basic familiarity with model checking.

Model checking algorithms vary with the logic of properties and the representation of state spaces. Our work views features as cross-products of of state machines; we extract properties of a feature by capturing the properties that are true at its boundary (interface) states. Capturing properties at states requires states to be labeled with individual properties. 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 spaces symbolically rather than explicitly [33]. Our algorithms are insensitive to this difference; indeed, we performed the verification tasks in this paper on a model checker employing symbolic representations [43].

Several researchers have described techniques for modular verification of designs [17, 22, 28, 36]. 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 techniques do not apply to sequential compositions, which frequently arise in software.

Some preliminary research [2, 11, 30] has begun to consider modular model checking with sequential, rather than parallel, control flow. The original work [30] handles designs with only one state machine; it also lacks a design framework, such as feature-oriented design, that drives the decomposition of the design. Subsequent work uses hierarchical state machines [2] and StateCharts [11] 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 features without knowing about all the other features that may exist. In other words, we need to verify *open*, rather than closed, systems.
- The designs include *multiple state machines per feature*, 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 (their 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 features that gather related sequential refinements into modules. Encapsulating related refinements in features allows us to verify properties of entire features in isolation from other features, even when those features cross-cut several actors. Without a feature-oriented architecture, isolating this information from across parallel modules is difficult if not impossible.

Lam and Shankar use *projections* as a way of decomposing protocols around different functions or operations in the protocol [29]. The functions around which they decompose a protocol strongly resemble features. At a theoretical level, their work more closely resembles abstraction rather than modular verification: they reduce portions of the design that are not in the feature being verified to single nodes or small subgraphs, and argue correctness based on refinements of abstractions. This approach is largely *decompositional*: it aims to make verification tractable on large designs. Our work, in contrast, is *compositional*; we support building different systems from the same set of designer-provided modules. This distinction requires us to treat features as open systems, and requires some special techniques in our methodology (as presented in this paper). Our compositions do not map inherently to abstraction refinements.[1]

Inverardi, Wolf, and Yankelevich extract graph-based representations of software components for purposes of checking for deadlock-freedom [26]. Their work, like ours, also represents components by graph-based abstractions and traverses those graphs to check properties. Our work differs from theirs in two key ways: first, we are establishing an overall methodology for handling feature-oriented 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).

A good deal of software architecture research has focused on support for product-lines and separation of concerns. Some of this work uses *layered architectures* [38], whose high-level structure resembles feature-oriented 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 features. 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 [37]. Their work did not consider formal analyses of designs under orthogonal architectures. Griswold and Notkin explore performance issues arising from layered architectures [21]. While their layers resemble features, 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 feature. In general, the loose coupling in feature-oriented architectures occurs between, rather than within, the "layers".

Several recent efforts have applied model checking to reasoning about aspect-like constructs. Chechik and Easterbrook reason about compositions of concerns using multi-valued model checking [8]. Their framework helps identify which concern (feature) is responsible for property violations when checking composed systems; it does not address how to prove properties through compositional reasoning on individual concerns. Ubayashi and Tamai propose a method of applying model checking to programs written using AspectJ [44]. Their verification methodology, however, is simply to weave the program together and verify the result. They leave any notion of compositional verification to future work. Nelson, Cowan and Alencar [34] present a case study on reasoning about cross-cutting concerns using

[1] Whether a more complicated mapping exists is an open question.

Alloy and the LTS checker. They too, however, compose the concerns into a single global specification in lieu of defining a compositional verification model. Lin and Lin also present a temporal logic-based approach to reasoning about feature interactions, but their approach is not compositional [32].

## 3 Verifying Feature-Oriented Software Designs

### 3.1 A Model of Feature-Oriented Design

We view a design as a set of classes, roughly one per actor in the design. A feature consists of a set of class extensions (*i.e. mixins* [7, 20, 39, 40, 46]) for the actor classes; the set of mixins in a feature relate to a common task 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 [13, 14, 26]. As FSATSis 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 features; 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 existing designs 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 features; a large number of interfaces might require too much overhead in our methodology.

Figures 3 and 7 show examples of base designs, features, extensions, and interfaces; Sections 3.3 and 3.4 explain the examples in detail. The following formal definition makes our model of feature-oriented 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, $\Sigma$ is the input alphabet, $\Delta$ 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

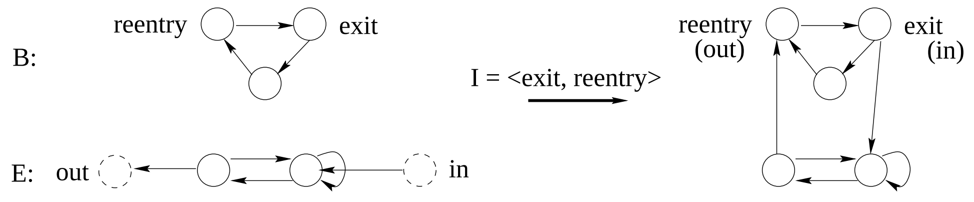


Figure 2: Composition of a base system $B$ with an extension $E$ via an interface.

set of propositional logic expressions over $\Sigma$), and $L : S \to 2^{\Delta}$ indicates which output propositions 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

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

two sets of formulas called *ExitLabels* and *ReentryLabels*, and a predicate *UntilCheck* on CTL formulas.[2] 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. Algorithm 1 explains the contents of *ExitLabels*, *ReentryLabels*, and *UntilCheck*.

**Definition 3** An *extension* is a tuple $\langle E_1, \ldots, E_n \rangle$ of state machines. Each $E_i$ must be 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 feature 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 2. 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 sets

[2]We will expand the definition of interfaces in Section 3.4.5.

of states in the constituent machines of base systems and extensions are disjoint.

**Definition 4** Composing base system $B = \langle M_1, \ldots, M_k \rangle$ and extension feature $E = \langle E_1, \ldots, E_k \rangle$ 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 $M_i = \langle S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s_{0_{Mi}}, R_{Mi}, L_{Mi} \rangle$ and its corresponding extension $E_i = \langle S_{Ei}, \Sigma_{Ei}, \Delta_{Ei}, s_{0_{Ei}}, R_{Ei}, L_{Ei} \rangle$ as follows: $S_{Ci} = S_{Mi} \cup S_{Ei} - \{in_i, out_i\}$; $s_{0_{Ci}} = s_{0_{Mi}}$; $R_{Ci}$ 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 feature-oriented designs. Specifically, our methodology supports the following activities:

1. Proving a CTL property of an individual feature or composition of features.[3] This is easily done in the base system with existing techniques, but becomes more complicated in extension features when a system has multiple actors.

2. Deriving a set of constraints on the exit and reentry states of a feature that are sufficient to preserve a particular property after composition (the *preservation constraints*).

3. Proving that a feature satisfies the preservation constraints of another feature (or existing system). This activity is only meaningful if the preservation constraints were generated for the exit and reentry states to which the new feature will attach. We establish preservation by analyzing only the extension, not the composition of the extension and the existing system.

[3]Clarke, Grumberg, and Peled's text [10] provides an overview of CTL and its use in verification. Our work uses negation normal form, which requires the CTL *release* operator: A[$\phi$ R $\psi$] is true if on all paths, $\psi$ is true on all states until $\phi$ is true, but $\phi$ is not required to eventually be true (in other words, R is a *weak* operator, whereas U is usually a strong operator).

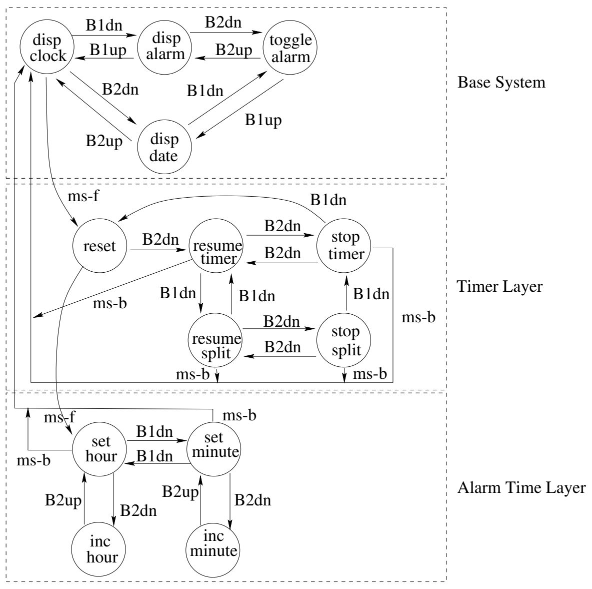


Figure 3: A collaborative design for a sportswatch.

These activities correspond to a kind of modular verification, where the features 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 feature therefore contains and extends only one state machine. This example motivates our interfaces and high-level approach to sequential feature composition. The communication protocol captures key characteristics of FSATSand shows how our methodology extends to designs with multiple state machines in each feature. Formal presentations of our algorithms and their correctness proofs follow each motivating example. Section 3.5 discusses some pragmatic issues that arise when performing these verification runs with an existing model checker.

## 3.3 Single-Machine Designs

Figure 3 shows a feature-oriented 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 feature also supports a split timer for capturing time instantaneously. The second extension supports setting the alarm time; we omit features for setting the clock time due to space constraints. Although both extensions add core functions, rather than optional features, we implement them as features 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 *disp-clock* in CTL). This property is easy to verify using a model checker. The base feature publishes one interface: $\langle dispclock, dispclock\rangle$, 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 feature—to fail. We could compose the base system and timer features 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 feature; 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 feature will preserve the property already proven of the base system without using the entire base system.

The classical CTL model checking algorithm [9] 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 feature states do so through the exit state.

Given the base system interface ($\langle dispclock, dispclock\rangle$ in this case) and a property to preserve (AG EF *dispclock*), 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*:

- *dispclock*
- EF*dispclock*

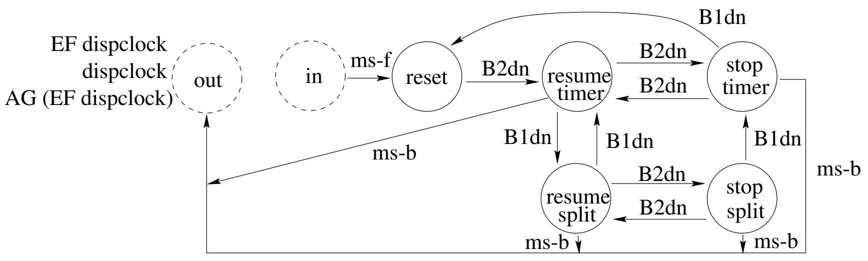


Figure 4: The timer extension with marking assumptions on the *out* state.

- AG(EF*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 assumes 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. Figure 4 shows the sportswatch timer feature 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. Markings of existential properties are guaranteed to hold since composition only ever adds states; we therefore only need to check preservation of non-existential properties.

In addition to the display-clock property, we can also verify that the timer feature (without the base feature attached) satisfies the property "once started, the timer can always be stopped" (AG(*resumetimer* $\rightarrow$ EF *stoptimer*)). We view the timer feature as the base system and the base as the extension to verify that the base feature would preserve this property upon composition. In general, given two features, we can view each as an extension to the other, thus allowing us to check whether they preserve each others' properties upon composition.

We also construct a composed system from the base and timer features, with interface $\langle dispclock, reset\rangle$. The interface states change after composition because the watch requires switching between modes to be deterministic; satisfying this constraint requires new features to be entered from the timer feature, 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 feature and from verifying the preservation of the other feature's properties. For *dispclock*, the new set of interface markings is:

- *dispclock*;
- $\mathsf{EF}$*dispclock*;
- $\mathsf{AG}$($\mathsf{EF}$ *dispclock*);
- $\mathsf{EF}$*stoptimer*
- *resumetimer* $\rightarrow$ $\mathsf{EF}$*stoptimer*
- $\mathsf{AG}$(*resumetimer* $\rightarrow$ $\mathsf{EF}$ *stoptimer*)

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

## The Algorithms on Single State Machines

The discussion of the stopwatch example motivated two algorithms, one for model checking formulas of features to populate their interfaces and one for checking whether one feature preserves the properties of another. We present both algorithms here, followed by a theorem about the compositionality of our approach. The preservation algorithm confirms only that properties that label the interface state before composition are still valid after composition; additional properties may become true on the interface state after composition (but none would logically contradict an existing property label).

**Algorithm 1 (Populating Interfaces During Verification)** *Given a property $\varphi$ to check of a base system $B$:*

1. *Use standard CTL model checking to check $\varphi$ in the start state of $B$.*
2. *Store the labels on the interface states of $B$ in the interface.*
3. *For each subformula of $\varphi$ the form $\mathsf{A}[\phi \; \mathsf{U} \; \psi]$, check whether $\psi$ is true somewhere along every path from* reentry *to* exit *in $B$. This can be done by model checking formula $\mathsf{AG}$(reentry$\rightarrow$ $\mathsf{A}[!$exit $\mathsf{U} \; \psi])$ on $B$. If the formula fails, set UntilCheck($\mathsf{A}[\phi \; \mathsf{U} \; \psi]$) to true in the interface.*

**Algorithm 2 (Preserving Properties Upon Extension)** *Given an extension $E$ with placeholder states* in *and* out *and an interface $I$ for the machine with which $E$ will compose:*

1. *Initialize the labels on* in *to the set of all positive and negative propositions labeling* exit *in $B$.*

2. *For each formula $F$ that labels the state* reentry *in $I$, add $F$ as a label to state* out *in $E$ unless the following two conditions are all satisfied:*

   - *$F$ is of the form $\mathsf{A}[\phi\ \mathsf{U}\ \psi]$,*

   - UntilCheck*(F) is true in $I$ (meaning that $\psi$ is false at every state on some path from re-enter to exit in the original system).*

3. *For each non-propositional formula $F$ labeling the state* exit *in $I$:*

   - *If $F$ is of the form $\mathsf{A}[\phi\ \mathsf{R}\ \psi]$, $F$ does not label* reentry *in $I$, and $\psi$ does label* reentry *in $I$, then model check $\mathsf{A}[\psi\ \mathsf{U}\ \phi]$ at state* in *in $E$. If the check succeeds, then add $F$ as a label to* in.

   - *If $F$'s topmost operator is an existential temporal operator, then add $F$ as a label to* in *without model checking.*

   - *Otherwise, model check $F$ at state* in *using the standard CTL model checking algorithm.*

   *We must iterate over the labels in order of increasing length; the ensures that formulas are checked after their subformulas are checked. The correct operation of the original CTL model checking algorithm depends on this ordering restriction.*

4. *If all labels on* exit *are labels of* in *then report that composition preserves all properties of the base system. If some label on* exit *is not a label on* in, *then report that composition may violate some properties of the base system.*

Laster and Grumberg independently derived a similar algorithm for reasoning about sequential decomposition of finite state machines [30]. Their algorithm is specified on program constructs (such as **if** statements and **while** loops); the details are sufficiently different that it does not carry over to our context. As a result, we provide our own proof of correctness in this paper. The 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). We present the correctness argument for the single-machine case here, and build the result for the multiple-machine case upon this proof in Section 3.4.4. The argument consists of two theorems that we prove correct through a single inductive proof with the following structure. (The vertical ellipses denote induction. The solid arrows chart a path through the proof. At each stage, we first prove (or assume by induction) the proposition at the tail of a solid arrow. Using its veracity, we prove the proposition at the head of the arrow.)

$$\begin{tikzcd}
\text{Theorem 1} & \text{Theorem 2} \\
|\varphi| = 1 & |\varphi| = 1 \\
\vdots & \vdots \\
|\varphi| <= k \arrow[r] & |\varphi| <= k \arrow[dl] \\
|\varphi| <= k+1 &
\end{tikzcd}$$

**Theorem 1** *Let $B$ be a base machine, $I$ be an interface containing states* exit *and* reentry*, and $E$ be an extension with placeholder states* in *and* out*. Let $C$ be the system composed from $B$ and $E$ via $I$. If Algorithm 2 claims that all properties of $B$ are preserved under composition of $B$ and $E$ via $I$, then all labels on* exit *in $B$ are true at* exit *in $C$.*

**Theorem 2** *Let $B$ be a base machine, $I$ be an interface containing states* exit *and* reentry*, $\varphi$ be a property in negation normal form that labels a state $s$ in $B$, and $E$ be an extension with placeholder states* in *and* out*. Let $C$ be the system composed from $B$ and $E$ via $I$. If Algorithm 2 claims that all properties of $B$ are preserved under composition of $B$ and $E$ via $I$, then $\varphi$ is true at $s$ in $C$.*

**Proof:** The proofs are by induction on the structure of CTL formulas. In the base case, the formula is a positive or negative atomic proposition. By construction (Defn 4), $C$ copies the values of atomic propositions from $B$ and $E$ to all states. $B$ and $E$ share only the interface states, and they also have the same sets of atomic propositions by construction (Algorithm 2, steps 1 and 2). Both theorems are therefore true in the base case.

Assume that Theorem 1 is true of all formulas of length no more than $k$ and that Theorem 2 is true of all formulas of less strictly less than $k$. We first prove that Theorem 2 is also true of all formulas of length $k$. By definition, a CTL model checker determines the labels at a state from the labels of its successor states [10]. Let $s$ be a state in $B$. All labels on $s$ in $B$ will be true at $s$ in $C$ whenever $B$ and $C$ agree on both the successor states of $s$ (transitively closed) and the labels on those states.

By construction (Defn 4), $B$ and $C$ can only disagree on the successors of $s$ if $s$ reaches *exit* in $B$. If $s$ is *exit*, then all labels of length $k$ in $B$ also hold at $s$ in $C$ by the inductive assumption on Theorem 1. Assume $s$ is not *exit*. For all states $s'$ on the path from $s$ to *exit*, all labels on $s'$ in $B$ of length less than $k$ must hold of $s'$ in $C$ by the inductive assumption on Theorem 2. Since labels of length up to and including $k$ on *exit* carry over from $B$ to $C$ and all shorter labels hold at each $s'$, a simple inductive argument establishes that all labels on $s$ in $B$ are also true on $s$ in $C$.

We now use both the inductive assumption on Theorem 1 and the result that Theorem 2 holds for all formulas of length no more than $k$ to prove that Theorem 1 holds for formulas of length $k+1$. This will be sufficient to establish the truth of both theorems.

Assume Algorithm 2 claims that all properties of $B$ are preserved upon composition with $E$. By definition, all labels on *exit* in $B$ are therefore also labels on *in* in $E$. Let $\varphi$ be a formula of length $k+1$ that labels both *exit* in $B$ and *in* in $E$. We must prove that $\varphi$ would also label *exit* in $C$. The argument depends on $\varphi$'s structure:

- If $\varphi$ is of the form $\phi \wedge \psi$, then both $\phi$ and $\psi$ must label both *exit* in $B$ and *in* in $E$. By the inductive hypothesis, both $\phi$ and $\psi$ label *exit* in $C$, so $\varphi$ must label *exit* in $C$.

- If $\varphi$ is of the form $\phi \vee \psi$, then one of $\phi$ or $\psi$ must label both *exit* in $B$ and *in* in $E$ (if $\phi$ labeled only one and $\psi$ only the other, Algorithm 2 would not claim that all labels are preserved on composition). By the inductive hypothesis, the same subformula must label *exit* in $C$, so $\varphi$ must label *exit* in $C$.

- If $\varphi$'s top-level operator is an existential temporal operator ($\mathsf{EX}$, $\mathsf{EU}$, $\mathsf{ER}$, or $\mathsf{EG}$), then $\varphi$ labels *exit* in $B$ because there exists a path from *exit* in $B$ that satisfies $\varphi$'s semantics. Since all paths in $B$ are also in $C$ by construction, and since $B$ and $C$ must agree on states satisfying subformulas of $\varphi$ by the inductive assumption on Theorem 2, $\varphi$ must also label *exit* in $C$.

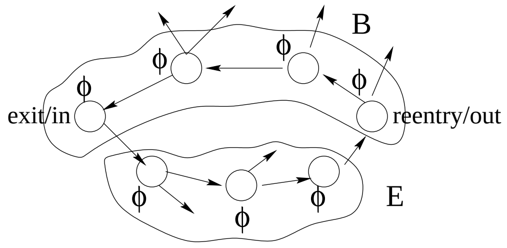
- If $\varphi$ is of the form $\mathsf{AX}\phi$, then $\varphi$ would be true at *exit* in $C$ if every successor of *exit* in $C$ satisfies $\phi$. By assumption, $\varphi$ labels both *exit* in $B$ and *in* in $E$; this means that $\phi$ is true at every successor of these states in their respective machines. By construction, every successor of *exit* in $C$ is a sucessor of either *exit* in $B$ or *in* in $E$. By the inductive assumption on Theorem 2, $\phi$ must be true of each successor from *exit* in $C$.

The inductive assumption for Theorem 2 states that all labels on *reentry* in $B$ must label *reentry/out* in $C$. Since *reentry* must lie on any path from $E$ back to $B$, it follows that all labels of length no more than $k$ on states in $E$ must be true at the corresponding states in $C$. Since $\phi$ is such a label, $\phi$ must be true of each successor of *in* in $C$. Since all successors of $C$ satisfy $\phi$, $\varphi$ must label *exit* in $C$ according to the definition of $\mathsf{AX}$.

- If $\varphi$ is of the form $\mathsf{A}[\phi \mathsf{U} \psi]$, then two conditions must hold for $\varphi$ to be true at *exit/in* in $C$: first, every path from *exit* in $C$ must eventually satisfy $\psi$, and second, every state along each path must satisfy $\phi$ until a state satisfying $\psi$ is reached. Since $\varphi$ labels *exit* in $B$ and *in* in $E$ (by definition), these two conditions hold of every path that stays entirely within one of $B$ or $E$. We therefore only need to consider paths that involve states of both $B$ and $E$ other than *exit* and *in*.

  Since the only transition in $C$ from a state in $B$ to a state in $E$ is the one leaving *exit/in*, we only need to consider paths that start at *exit/in* in $C$, pass through states in $E$, then continue with states in $B$.[4] By construction, all such paths must include the state *reentry/out* at the interface from $E$ back into $B$. Assume $P$ is such a path that does not satisfy $\varphi$ in $C$. One of the two conditions must fail to hold; we consider each case in turn.

  1. Assume $P$ contains no state that satisfies $\psi$. By assumption, $\varphi$ labeled *in* in $E$. Since the prefix of $P$ that passes through $E$ cannot contain a state that satisfies $\psi$, $\mathsf{A}[\phi \mathsf{U} \psi]$ must label *out* in $E$ (by the CTL semantics). By step 2 of Algorithm 2, $\mathsf{A}[\phi \mathsf{U} \psi]$ therefore labels *reentry* in $B$. This label implies that $\psi$ is eventually true on every path within $B$ that starts at *reentry* and includes no states from $E$. Therefore, $P$ must include the *exit* state from $B$. $P$ thus has a prefix with the following structure (modulo cycles within $B$ and $E$), where none of the states in $P$ is labeled with $\psi$:



     Since $\mathsf{A}[\phi \mathsf{U} \psi]$ labels *out* in $E$, *UntilCheck($\varphi$)* must have been false in $I$ (by Algorithm 2, step 2).

[4] Since CTL model checking uses backward propagation, labels on *exit/in* will change based on labels that propagate through $E$.

According to the definition of *UntilCheck*, $B$ therefore contained no path from *reentry* to *exit* on which $\psi$ was false in all states. $P$ contradicts this requirement, so no such $P$ can exist.

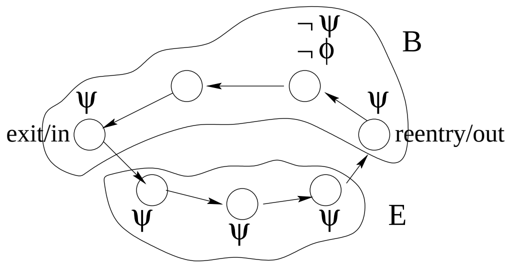
2. Assume $P$ contains a state at which $\phi$ is false before $\psi$ is true. Let $s'$ be the first such state in $P$. Since $\mathsf{A}[\phi\ \mathsf{U}\ \psi]$ labels *in* in $E$, $s'$ cannot lie in $E$. Therefore, $s'$ must lie in $B$, all states in $P$ that lie within $E$ must satisfy $\phi$, and $\mathsf{A}[\phi\ \mathsf{U}\ \psi]$ must label *out* in $E$ (and hence *reentry* in $B$ by step 2 of Algorithm 2). $B$ therefore contains a path from *reentry* to $s'$ that violates the label $\mathsf{A}[\phi\ \mathsf{U}\ \psi]$ on *reentry*. As this contradicts the CTL semantics, the path $P$ cannot exist.

These arguments shows that all paths involving states of both $B$ and $E$ must satisfy $\varphi$ in $C$.

- If $\varphi$ is of the form $\mathsf{A}[\phi\ \mathsf{R}\ \psi]$, then $\varphi$ will be true at *exit* in $C$ if every path starting from *exit* in $C$ has $\psi$ true at every state until some state satisfies $\phi$; unlike with $\mathsf{AU}$, $\phi$ is not required to be true at some state. Since $\varphi$ labels *exit* in $B$ and *in* in $E$, $\varphi$ must be true on all paths that lie entirely within $B$ or $E$. We therefore only need to consider paths that start at *exit/in* in $C$, pass through states in $E$, then continue with states in $B$.

  Assume there exists such a path $P$ that does not satisfy $\varphi$ in $C$. Let $s'$ be the first state in $P$ at which $\psi$ is false before $\phi$ is true. Since $\varphi$ labels *in* in $E$, $s'$ cannot lie in $E$ and either $\varphi$ or $\psi$ must label *out* in $E$ (by the CTL semantics). $P$ would satisfy $\varphi$ if $\varphi$ labels *reentry/out* in $C$ (by the CTL semantics), so *reentry/out* must have label $\psi$ and not label $\varphi$.

  Since $\psi$ labels *reentry* in $B$ while $\varphi$ does not, then the suffix of $P$ from *reentry* must contain a state on which $\psi$ is false before $\phi$ is true. Thus, $P$ must have the following structure (modulo cycles):



  Notice that the prefix of $P$ that lies in $E$ does not satisfy the formula $\mathsf{A}[\psi\ \mathsf{U}\ \phi]$. According to step 3 of Algorithm 2, however, the labels on *reentry* in $B$ required that $\mathsf{A}[\psi\ \mathsf{U}\ \phi]$ be true at *in* in $E$ in order for $\varphi$ to label *in*. This contradicts the premise that $\varphi$ labels *in* in $E$, so $P$ cannot exist.

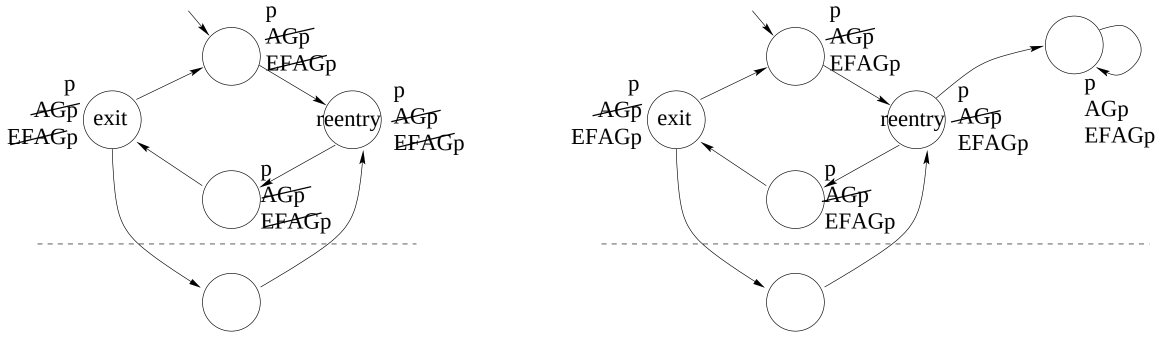


Figure 5: Two cases in which a label that was on *exit* in $B$ is not a label on *in* in $E$. The portion above the dashed lines lies in $B$ and the portion under lies in $E$. In both examples, the crossed-out formulas hold in $B$ but fail to hold in $C$; the labels in boxes are the ones that fail to hold on *in* in $E$. In the example on the left, the change in label causes the $\mathsf{AG}\ \mathsf{EF}p$ to be violated in the start state; in the example on the right, the $\mathsf{AG}\ \mathsf{EF}p$ continues to hold in the start state despite the change in labels.

- If $\varphi$ is of the form $\mathsf{AG}\phi$, then $\varphi$ would be true at *exit/in* in $C$ if $\phi$ were true at every state reachable from *exit/in* in $C$. Since $\varphi$ labels both *exit* in $B$ and *in* in $E$, $\phi$ must be true at every state reachable from *exit* in $B$ and *in* in $E$. By the inductive assumption on Theorem 2, $\phi$ must be true at every state in $C$ from $B$. Further, it must be true at every state in $C$ from $E$ since the preservation step determined that $\varphi$ is true at *in* in $E$. $\varphi$ is therefore true at *exit/in* in $C$.

□

The proof justifies the decision in step 3 of Algorithm 2 to not check preservation of existential formulas in $E$. Our implementation utilizes this optimization.

Theorem 2 proves that Algorithm 2 is sound; it is not, however, complete. The algorithm reports a potential violation whenever a label on the *exit* state in $B$ fails to hold on the *in* state of $E$. There is no guarantee, however, that the failed label is actually critical to the truth of a property on the base or composed system. Figure 5 shows two examples for which Algorithm 2 reports a potential violation; in one case the desired property fails while in the other it continues to hold. In theory, we could reduce the number of violation warnings in two ways: we could attempt to compute some additional information in the base system about which overall properties depend on which labels of *exit* and *reentry*, or we could require all previously checked labels that were false at *exit* in $B$ to remain false in *in* in $E$. The former would require more interface information, while the latter could be overly restrictive and require too much computational overhead for insufficient gain. Additional experience with our original algorithm will guide the degree

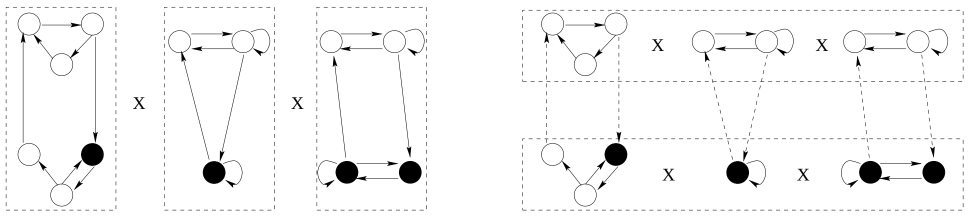


Figure 6: Actor-oriented versus feature-oriented decompositions: a state-machine based view.

to which we need to reduce the numbers of reported false negatives.

**Implementation:** For our experiments, we simulated Algorithm 2 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 FSATSbecause FSATShas multiple state machines in each feature. In practice, almost all interesting feature-oriented designs, by their very nature, will employ multiple state machines (one for each actor). When each feature contains a single state machine, extending a design with a feature corresponds to sequential composition of state machines. When features contain multiple state machines, extending a design with a feature corresponds to a hybrid of sequential and parallel composition: the machines within a feature are composed in parallel (because they run together to implement a particular feature), but the features 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 indepen-

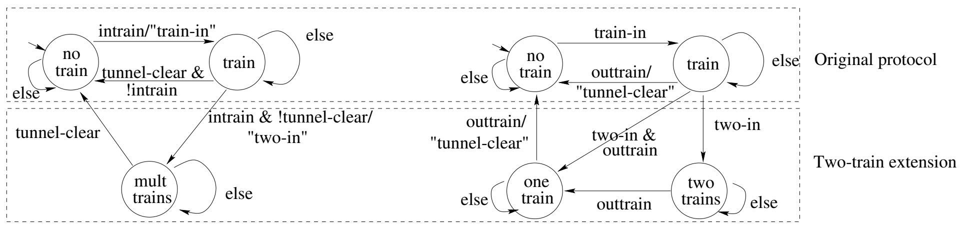


Figure 7: A feature-oriented design for a track-operator communication protocol.

dent verification of features. Figure 6 (right) 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 6 (left) 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 feature-oriented 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 feature-oriented verification of designs with multiple state machines per feature. We must construct the parallel compositions representing each feature 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 features. Our algorithm is designed to create parallel compositions that can in turn be composed sequentially with other features. We describe the algorithm by illustrating its behavior on a small example. We also evaluate this algorithm's ability to verify properties of features in isolation. While many features (including the FSATSfeatures) 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 features. 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 FSATSwithout necessitating as much explanation of the domain.

### 3.4.1 The Clayton Tunnel Protocol

We consider a feature-oriented design of a communications protocol between operators at either end of a train tunnel (see Figure 7) [25]. 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 feature, 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 feature 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 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 feature, we would like to prove that the inbound operator never

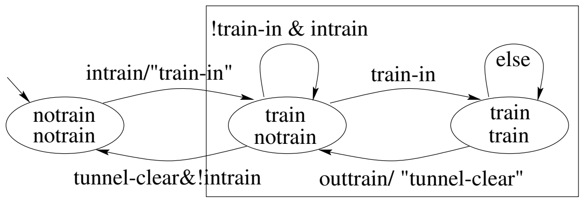


Figure 8: The cross-product state machine (reachable subset) for the tunnel base feature. The exit subgraph for an interface containing both *train* states as exit states is enclosed in the solid box. Both states in the exit subgraph have the potential to transition to an extension.

livelocks thinking that there is a train in the tunnel (AG(EF(*inbb.state=notrain*))); this property requires all trains in the tunnel to eventually exit the tunnel, which we handle with a fairness constraint [10]. 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 (AG ((*inbmsg=two-in*)$\rightarrow$ A(!(*instate=out*) U (*outbmsg=tunnel-clear*)))).

### 3.4.2 Creating the Extension Cross-Product

The extension consists of the two state machines in the lower dashed box in Figure 7 (though with *in* and *out* states, as in Figure 4). 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 [10]. 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.

This synchronization observation reflects a general technical challenge with feature-oriented 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 feature-oriented designs roughly synchronize actors around each feature, these *hybrid* states can arise in two

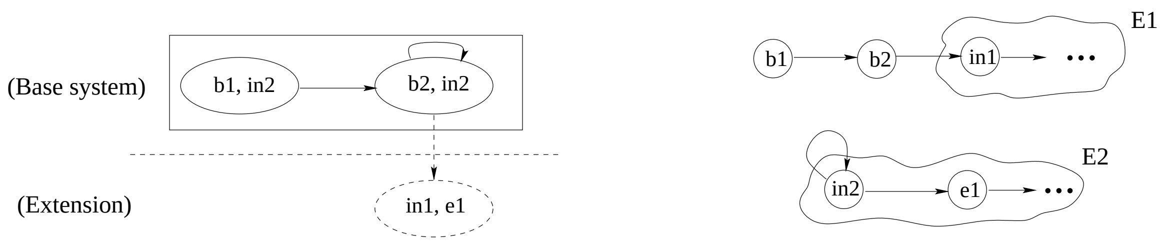


Figure 9: Decomposing an exit subgraph to drive extension cross-product construction.

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 8 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 feature. 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. Not all states in the exit subgraph will, however, *necessarily* have a transition into the extension feature; in fact, the composed machine may well contain paths through the exit subgraph that eventually lead to the feature extension. Constructing the extension cross-product correctly therefore requires that we account for these potential paths. Our methodology therefore prepends each individual machine in the extension with states from the exit subgraph prior to cross-product construction.

Figure 9 illustrates the needed construction. The diagram on the left shows an exit subgraph and a potential state ($\langle in_1, e_1 \rangle$) 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 ($\langle b_2, in_2 \rangle$), 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 $ES$ (by Definition 5).

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.

This construction yields a set of cross-product states and transitions involving states from the original extension machine $E_i$; some of these states lie entirely within the extension, while others are hybrid states. The composed

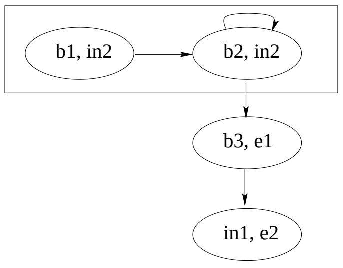


Figure 10: Using escape states to detect insufficient synchronization of interface states.

cross-product machine arising from Definition 4 (parallel composition of complete actors) also yields a set of cross-product states and transitions involving states from the original extension machine $E_i$. Our extension cross-product construction is correct if these two sets of states and transitions are identical. The correctness proof (presented formally at the end of this section) 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 10. 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

Given the parallel composition of the extension machines constructed using the exit subgraph, we can attempt to verify the feature 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 environment models with extensions 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 feature. 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 feature-oriented design. feature-oriented designs offer the hope of testing features in isolation. That testing, however, requires knowledge about the environment that will drive the feature. Our approach merely formalizes the problem of obtaining a restricted testing harness for feature-oriented 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 features independently from their base systems: capturing global states that bridge features 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 feature with a base system can be verified via sequential composition, rather than on the global composed state machine. Verifying multiple-machine features 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 features. 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 feature 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 feature-oriented verification methodology.

Fortunately, the feature-oriented 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 feature-oriented 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.[5]

The following constraints indicate when we can prove that a feature 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.

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 feature than when entering it. This seems necessary to avoid generating additional reachable states in the global base feature. 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

[5]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. Our methodology does support parallel decomposition, though. One can build the state machine for an entire individual actor by linking all of the state machine fragments for that actor vertically, as shown in Figure 6 (left). Our methodology enables this construction because each extension is a tuple of independent state machine fragments.

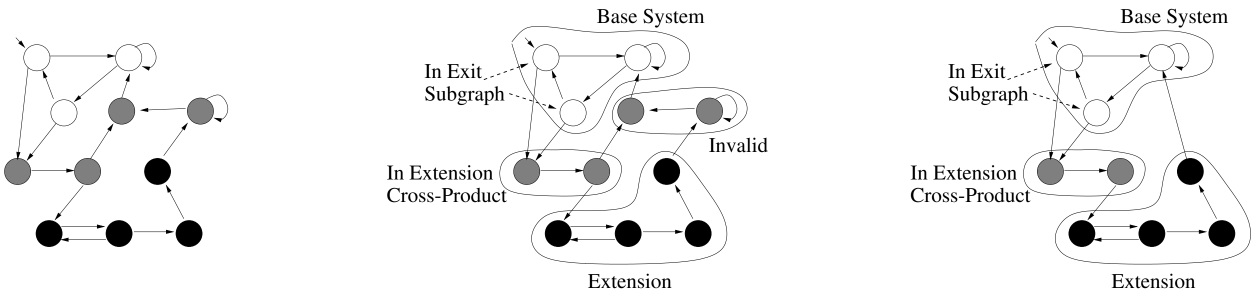


Figure 11: How the two state space constructions correspond. The graph depicts a global state space constructed over multiple actors. White states lie fully in the base system, black states lie fully in the extension, and gray states are hybrids. The annotated version in the middle shows which part of the construction covers each state. The restrictions on the state space would reject a system containing the states marked *invalid*. A valid system appears on the right.

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.

The given constraints are sufficient to establish a correspondence between the two constructions of the global composed state space. Figure 11 illustrates the intuition underlying the proof. The proof establishes that all reachable states under the standard parallel construction lie in either the base system (the white states in the figure) or the extension cross-product (the gray and black states in the figure). The interesting cases involve the hybrid states; the restrictions limit these to lying at feature entry.

**Theorem 3** *Let $B = \langle M_1, \ldots, M_k \rangle$ be a base system and let $I$=$\langle \langle exit_1, reentry_1 \rangle, \ldots, \langle exit_k, reentry_k \rangle \rangle$ be an interface of $B$. Let $E = \langle E_1, \ldots, E_k \rangle$ be an extension feature. Let $G_1$ be the global composed state machine over $B$, $I$, and $E$ (Defn 4). Let $G_2$ be the state machine consisting of the base cross-product and the extension cross-product for $B$, $I$, and $E$ (in which states common to both cross-products are unified). If the methodology restrictions apply, then $G_1$ is isomorphic to $G_2$.*

**Proof:** We will establish isomorphism by mapping each state in $G_1$ to the state in $G_2$ with the same individual state components from the base and extension machines.

Consider the subgraph of $G_1$ lying completely in the base system and the base system cross-product (a subgraph of $G_2$). Every state in the base system cross-product must lie in $G_1$ because the base cross-product arises from the same construction algorithm and initial states as the construction of $G_1$. By similar reasoning, every base system state in $G_1$ that becomes reachable before reaching a state involving the extension must lie in $G_2$. The subgraphs of $G_1$ and

$G_2$ over the base states are therefore isomorphic unless passing through the extension in $G_1$ makes some additional base system states reachable.

Under the restrictions on the designs to which our methodology applies, passing through the extension in $G_1$ cannot make any new states reachable. The restrictions require the tuple of reentry states to be reachable in the base system alone; this base system state is therefore in both $G_1$ and $G_2$. Any state reachable from the reentry-tuple state must also be in $G_1$ and $G_2$ by the definition of reachability. The restrictions also require the reentry tuple (the tuple of out states) to be reachable in the extension, and for all actors to stutter in their out states until all actors reach their out states. These restrictions guarantee that no hybrid states are reachable while passing from the extension back to the base system. Thus, no base system states are reachable from the extension unless they were reachable in the base-system cross product.

The transitions between the base system states are isomorphic in $G_1$ and $G_2$ by construction, since both the states and the transitions arise from the same construction algorithm.

Having shown that the states lying entirely within the base system are isomorphic between $G_1$ and $G_2$, it remains to show that the hybrid and extension states are isomorphic. We consider the hybrid states first. The exit subgraph is, by definition (Defn 5), the subset of the base cross-product consisting of all states at the interface to the extension. The reachable hybrid states in $G_2$ are those reachable from the exit subgraph states. Since all exit subgraph states are in $G_1$ by definition, all hybrid states reachable in $G_2$ must therefore be reachable in $G_1$.

We now show that all hybrid states reachable in $G_1$ are reachable in $G_2$. Assume $G_1$ contains a reachable state $s_1$ that is not reachable in the extension cross product of $G_2$; by assumption, $s_1$ is not in the exit subgraph. Assume $s_1$ has at least one predecessor $p_1$ that is reachable in both $G_1$ and $G_2$ (such an $s_1$ must exist). The extension cross product is constructed from the extension machines and the expanded states in the exit subgraph construction. Since all of the expanded states in this construction are also in $G_1$, then $s_1$ must have arisen from states not in the extension machines: in other words, $s_1$ must contain at least one base machine state as a component. Call this state $b_1$. Let $p$ refer to the component of cross-product state $p_1$ from the same actor machine as $b_1$. Then the transition from $p$ to $b_1$ would have yielded a transition to the escape state in the extension cross-product construction (since $b_1$ is not in an extension machine). $B$, $E$, and $I$ therefore would violate the constraints of our methodology, which require no escape states to be reachable in the extension cross product. By contradiction, then, the hybrid states in $G_1$ must all be reachable in $G_2$.

This leaves only the extension states to compare between $G_1$ and $G_2$. Since these are reachable from the exit subgraph and/or hybrid states, all of which are already common to $G_1$ and $G_2$, these states must be equivalent in the two graphs. $G_1$ and $G_2$ therefore have the same sets of reachable states under a well-defined mapping, and are hence isomorphic.

$\square$

**Lemma 1** *A CTL property is true of a global-composed state space iff it holds under our methodology.*

**Proof:** Theorem 3 argues that the base system plus the extension cross-product is isomorphic to the global-composed system of a given base system and extension. Our methodology therefore visits all of the states in the global-composed system during verification (since it visits all states in both the base and in the extension cross-product). The composition of the base system and extension cross-product is a sequential composition, so Theorem 2 is sufficient to prove that the verification is consistent with that on the global-composed system.

$\square$

### 3.4.5 Revising the Interfaces

In the multiple machine case, we needed the exit subgraph to perform verification compositionally. In addition, we needed to remember property labels on states of the cross-products of features, rather than on the interface states of individual features. The following definition formally extends the original definition of feature interfaces (Defn 2) to account for these changes.

**Definition 6** *(Interfaces, revised)* Given a feature (base system or extension) $\langle M_1, \ldots, M_k \rangle$, an interface for the feature contains

- a sequence of pairs of states $\langle \langle exit_1, reentry_1 \rangle, \ldots, \langle exit_k, reentry_k \rangle \rangle$ (where each $exit_i$ and $reentry_i$ is a state in machine $M_i$),
- the exit subgraph for the feature (as defined in Defn 5),
- a set of formulas *ExitLabels* on the global state $reentry_1 \times \ldots \times reentry_k$,

- a set of formulas *ReentryLabels* for each state in the exit subgraph, and
- a predicate *UntilCheck* on CTL formulas (defined as before).

### 3.5 Implementation

We have conducted all the model checking tasks described in this paper using the VIS model checker [43]. 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 features. 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 the extension features 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 feature 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 feature-oriented 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 feature-oriented software designs inhibit straightforward attempts at modular verification. We demonstrate that the standard decompositions of designs into parallel or sequential modules are insufficient for feature-oriented 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 features. Our technique can potentially apply to a variety of system designs including certain uses of components and aspects. It is especially applicable to feature-oriented product line designs.

We have concentrated solely on model checking because we want to understand the strengths and limitations of algorithmic verification on feature-oriented 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 FSATSsuite. A related question is how to extend our approach to handle LTL formulas. Our approach relies heavily on the state-based semantics of CTL, and hence does not automatically carry over to LTL's path-based semantics.

Within the realm of algorithmic verification, model checking may be overkill for verifying certain feature 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.

Feature-oriented designs can benefit from a broader scope of verification techniques. Early work on analyzing dependencies between features [3] must be formalized and incorporated into any validation framework. Addressing this problem may involve creating special architectural description languages for feature-oriented designs that capture these dependencies and simplify construction of the exit and reentry subgraphs. Finally, we have encountered feature-oriented 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 feature-oriented designs, Jia Liu for sharing his design for the sportswatch, Harry Li for useful comments on drafts, and the Automated Software Engineering Group at NASA Ames for engaging exchanges.

## 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. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, April 2002.

[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] M. Chechik and S. Easterbrook. Reasoning about compositions of concerns. In *Proceedings of the ICSE Workshop on Advanced Separation of Concerns*, May 2001.

[9] 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.

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

[11] 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.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] 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.

[14] 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.

[15] 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*, 12(2):159–182, 2002.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *IEEE International Conference on Computer-Aided Design*, pages 245–252, 2000.

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

[26] 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.

[27] 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.

[28] 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.

[29] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.

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

[31] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Mar. 1999.

[32] F. J. Lin and Y.-J. Lin. A building block approach to detecting and resolving feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 86–109. IOS Press, 1994.

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

[34] T. Nelson, D. D. Cowan, and P. S. C. Alencar. Supporting formal verification of crosscutting concerns. In *Reflection*, pages 153–169, 2001.

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

[36] 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.

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

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

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

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

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

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

[43] 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.

[44] N. Ubayashi and T. Tamai. Aspect oriented programming with model checking. In *International Conference on Aspect-Oriented Software Development*, Apr. 2002.

[45] 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.

[46] 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.