Presented at the Workshop on Engineering Automation for Software Intensive System Integration, Monterrey, California, June 18-22, 2001.

# A Model Checking Framework for Layered Command and Control Software

Kathi Fisler
Department of Computer Science
Worcester Polytechnic Institute
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Computer Science Department
Brown University
sk@cs.brown.edu

Don Batory
Department of Computer Science
University of Texas at Austin
dsb@cs.utexas.edu

Jia Liu
Department of Computer Science
University of Texas at Austin
jliu@cs.utexas.edu

March 21, 2001

**Abstract**

Most existing modular model checking techniques betray their hardware roots: they assume that modules compose in parallel. In contrast, layered software systems, which have proven successful in many domains, are really quasi-sequential compositions of parallel compositions. Most such systems demand and inspire new modular verification techniques. This paper presents algorithms that exploit a layered (or feature-based) decomposition to drive verification. Our technique can verify most properties locally within a layer; we also characterize when a global state space construction is unavoidable. This work is motivated by our efforts to verify a military fire simulation and support software system called FSATS.

## 1 Introduction

Today's software applications are modularized around objects that collaboratively interact to provide the functionality of an application. This is the fundamental starting-point for contemporary object-oriented design as well as contemporary modular model checking techniques [15, 19, 23, 27].

An alternate form of modularity centers around *features* rather than objects. Programmers design and construct applications by introducing one feature at a time. Each feature adds new capabilities and responsibilities to previously existing objects and introduces new objects to a design. A characteristic of features is that they are largely independent: this substantially reduces application complexity (because the concerns and implementation details of one feature are separable from those of others) and increases application extensibility (because new features can be easily added and unwanted features removed). Feature-oriented design is a form of step-wise refinement in which the refinements are entire features, rather than low-level changes to individual statements.

Many research efforts now approach design through features, including layers [5], collaborations [26], aspects [22] and units [14]. In this paper, we call them *layers* to evoke the visual imagery of feature-based refinement. Layers have been particularly successful in software product-lines, where each application of a product-line is defined by a unique combination of features. A brief sampling of successful designs in this vein includes a military command-and-control scenario simulator [4], a programming environment [13], network protocols and database systems [5, 6, 33], and verification tools [16, 30].

The success of layered designs at *implementing* software product-lines suggests a tantalizing prospect: they may also assist in *validating* product-lines. Layers have well-defined interfaces that permit their composition to build larger systems. Layers tend, at least in principle, to obey the characteristics of components [17, 20, 31], such as separate compilation, multiple instantiability and external linkage. Perhaps we can verify each layer individually, and perform cross-layer verification when composing layers.

As a case study, we are especially interested in a layered system called FSATS, which implements a military command-and-control scenario simulator [4]. In particular, we wish to apply model checking to FSATS. FSATS is a particularly good candidate for model checking for two reasons. First, the design includes specifications of state machines [4], which eliminates the problem of deriving such state machines from the software. Second, the system has several temporal properties (for instance, *every accepted mission eventually results in a weapon firing*) that are especially amenable to model checking.

FSATS is a complex collection of over 19 layers that can be composed independently to form scenario simulators. This immediately makes the straightforward application of model checking impossible, due the combinatorial number of possible systems, and the sizes of the larger compositions. FSATS has thus inspired our research into new forms of modular algorithmic verification. As FSATS is too complex to serve as a running example in this paper, we discuss it briefly to elicit its key characteristics, then illustrate our development on two simple examples that distill these characteristics.

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

Model checking algorithms vary with the logic of properties. Our work extracts properties of layers by examining the labels on interface states. This assumes the model checker uses state labeling, which is the technique employed for branching-time temporal logics such as CTL. To simplify the development, we present our algorithms assuming an explicit representation of the state space of a system. In practice, many model checkers represent state symbolically rather than explicitly [25]. Our algorithms are insensitive to this difference; indeed, we performed the verification tasks in this paper on a model checker employing symbolic representations [32].

Several researchers have described techniques for modular verification of designs [15, 19, 23, 27]. These techniques are based on a hardware-oriented notion of modularity, in which modules are composed in *parallel*. For instance, one module might be a CPU, while another module represents a floating-point co-processor. The research then shows how to ensure the preservation of individual properties about the CPU or floating-point processor; using these techniques to prove properties involving both devices requires substantial experience, and is not always possible. These results do not apply to most software systems, where control flows sequentially between modules.

Some preliminary research [2, 10, 24] has begun to consider modular verification with sequential, rather than parallel, control flow. The original work [24] handles systems with only one state machine; it also lacks a design framework, such as layered design, that drives the decomposition of the system. Subsequent work uses hierarchical state machines [2] and StateCharts [10] to provide this decomposition, but the resulting systems are still monolithic. In contrast, we analyze systems with three key distinguishing features:

- Each layer introduces a feature that was not previously present in the system; the layer does not simply refine existing features.
- Layers are developed without knowledge about all the other layers that may exist in a final, composed system.
- Each layers (unit of sequential composition) encapsulates simultaneous extensions to multiple state machines.

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 systems that involve quasi-sequential composition of parallel compositions, such as exist in FSATS. Alur *et al.* discuss analysis techniques for sequential refinements within modules that are composed in parallel (this

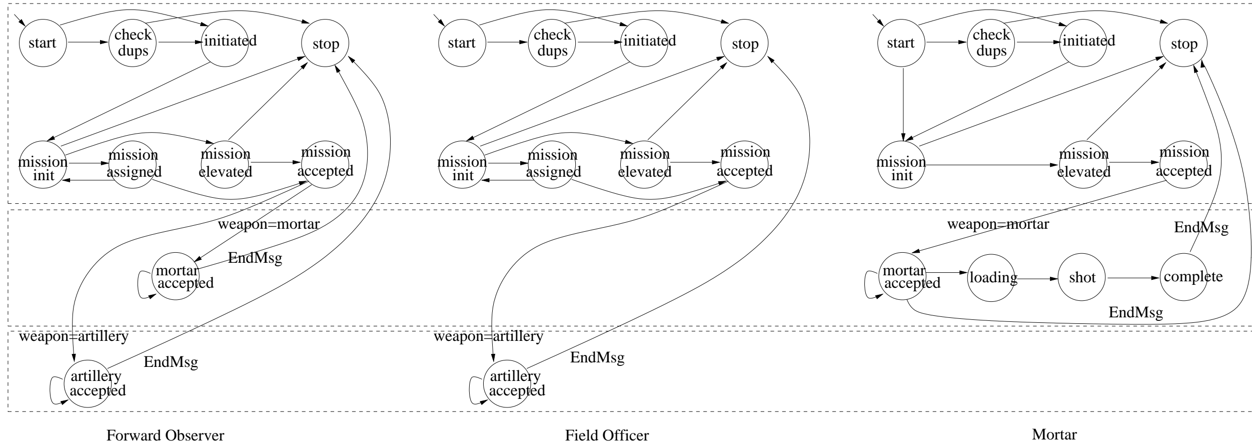


Figure 1: A sample of layered state machines from FSATS. The top layer is the base system, the second adds missions that fire mortars, and the third adds missions that fire artillery. The full design contains state machine hierarchies for the artillery as well as for other personnel in the command hierarchy.

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 layers that gather related sequential refinements into modules. Encapsulating related refinements in layers allows us to verify properties of entire features in isolation from other features, even when those features cross-cut several actors (*i.e.*, objects). Without a layered architecture, isolating this information from across parallel modules is difficult if not impossible.

# 3 Verifying Layered Software Systems

## 3.1 FSATS: An Example of Layered Design

FSATS is a command and control simulator. At core, it consists of a series of protocols for selecting weapons to fire at potential targets. The actors in FSATS are various military personnel (forward observers, field officers, brigade commanders, etc) arranged in a command hierarchy and the weapons at their disposal. Observers repeatedly identify potential targets and send messages along the command hierarchy to initiate missions against the targets. The personnel in the hierarchy accept or forward missions depending upon the weapons at their disposal. In the FSATS implementation, each potential target spawns a new thread in which to execute the protocol for handling that target.

Batory *et al.* have presented a layered design of FSATS, written largely as a set of layered state machines [4]. Figure 1 shows a sample of the layered state machines that comprise FSATS. A careful look at the machines and the code shows several characteristics that are potentially interesting from a model checking perspective:

- Each layer's (extension) state machines compose sequentially with their corresponding base machines.
- Each layer (extension) attaches to a common start and end point from the base layer.
- The conditions under which control enters a particular layer are similar across all actors and are closely coordinated through message passing.
- The state machines essentially synchronize at the end of a mission (on the EndMsg messages) right before the involved actor threads terminate.

These four observations drive the methodology and algorithms presented in this paper. The combination of these characteristics enable a powerful, modular approach to verification in which we verify layers individually and reason about property preservation under layer composition.

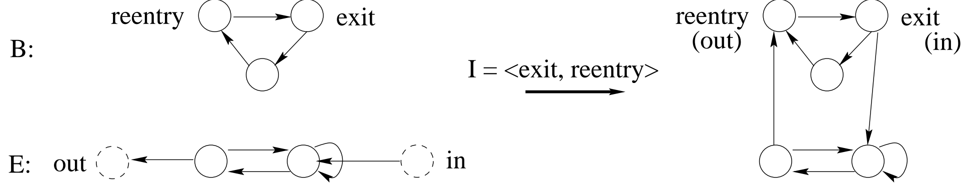


Figure 2: Composition of a base system $B$ with an extension $E$ via an interface.

## 3.2 A Model of Layered Design

We view a design as a set of classes, roughly one per actor in the system. A layer consists of a set of class extensions (*mixins* [7, 18, 28, 29, 34]) for the actor classes. The set of mixins in a layer relate to a common task, or *feature*, in the overall system (in FSATS, the features generally represent missions that utilize different weapons). This definition permits actor classes and mixins of arbitrary complexity. To make the problem of verification more tractable, we assume each actor class can be described as a state machine, and that each mixin extends an existing (base) state machine by adding nodes, edges, and/or paths between states in the base machine. State machine models of software arise from one of two sources: either the software is written in terms of state machines, as is true for many embedded software applications, or abstraction techniques derive state machines from the source code [11, 12]. FSATS is of the former flavor. Our work could adapt to the latter if the abstractions produce machines for which we could define meaningful interfaces between layers; accordingly, we regard the work on state machine abstractions as orthogonal to this paper.

Each base or composed system specifies interfaces, in terms of states, at which clients may attach extensions. We define interfaces formally below. In our experience, new features generally attach to the base system at common or predictable points, as Figure 1 illustrates; the set of interfaces is therefore small. This is important, as the interface states will indicate information that we must gather about a system in order to perform compositional verification of layers; a large number of interfaces might require too much overhead in our methodology.

Figures 3 and 6 show examples of base systems, layers, extensions, and interfaces; Sections 3.4 and 3.5 explain the examples in detail. The following formal definition makes our model of layered 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 set of propositional logic expressions over $\Sigma$), and $L : S \rightarrow 2^{\Delta}$ indicates which output symbols are true in each state.

**Definition 2** A *base system* consists of a tuple $\langle M_1, \ldots, M_k\rangle$ of state machines and a set of *interfaces*. We denote the elements of machine $M_i$ as $\langle S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s_{0_{Mi}}, R_{Mi}, L_{Mi}\rangle$. An interface contains a sequence of pairs of states $\langle\langle exit_1, reentry_1\rangle, \ldots, \langle exit_k, reentry_k\rangle\rangle$ Each $exit_i$ and $reentry_i$ is a state in machine $M_i$. State $exit_i$ is a state from which control can enter an extension machine, and $reentry_i$ is a state from which control returns to the base system. Interfaces also contain a set of properties and other information which are derived from the base system during verification; we describe these properties in detail in later sections.

**Definition 3** An *extension* is a tuple $\langle E_1, \ldots, E_n\rangle$ of state machines. Each $E_i$ must induce a connected graph, must have a single initial state with in-degree zero, and must have a single state with out-degree zero. For each $E_i$, we refer to the initial state as $in_i$ and the state with out-degree zero as $out_i$. States $in_i$ and $out_i$ serve as placeholders for the states to which the layer 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. The mortar mission in FSATS (Figure 1, first extension layer), for example, does not augment the protocol of field officers. We also assume that the states in the constituent machines of base systems and extensions are distinct.

**Definition 4** The composition of base system $B = \langle M_1, \dots, M_k \rangle$ and extension $E = \langle E_1, \dots, E_k \rangle$ via an interface $I{=}\langle\langle exit_1, reentry_1\rangle, \dots, \langle exit_k, reentry_k\rangle\rangle$ is a tuple $\langle C_1, \dots, C_k\rangle$ of state machines. Each $C_i = \langle S_{Ci}, \Sigma_{Ci}, \Delta_{Ci}, s_{0_{Ci}}, R_{Ci}, L_{Ci}\rangle$ is defined from $M_i = \langle S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s_{0_{Mi}}, R_{Mi}, L_{Mi}\rangle$ and $E_i = \langle S_{Ei}, \Sigma_{Ei}, \Delta_{Ei}, s_{0_{Ei}}, R_{Ei}, L_{Ei}\rangle$ 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 $C_1, \dots, C_k$ as the *global composed state machine*. Composed systems may serve as subsequent base systems by creating additional interfaces as necessary.

## 3.3 Verification Methodology

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

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

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

We illustrate our methodology using two examples: a sportswatch and a communication protocol. The sportswatch design consists of a single actor; each collaboration therefore contains and extends only one state machine. This example motivates our interfaces and high-level approach to sequential layer composition. The communication protocol captures the key characteristics of FSATS identified in Section 3.1 and shows how our methodology extends to designs with multiple state machines in each collaboration. We have performed all verification runs cited in these sections using the described methodology and the VIS model checker [32]. Section 3.6 discusses pragmatic issues behind these runs.

## 3.4 Single-Machine Designs

Figure 3 shows a layered 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 layer also supports a split timer for capturing time instantaneously. The second extension supports setting the alarm time; we omit layers for setting the clock time due to space constraints. Although both extensions add core functions, rather than optional features, we implement them as layers 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) 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 (written as AG EF *display_clock* in CTL). This property is easy to verify using a model checker. The base layer 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 layer — to fail. We could compose the base system and timer layers 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 layer; worse still, on a larger example, the composed design could be too large to model check. We therefore want to verify that the timer layer will preserve the property already proven of the base system without using the entire base system.

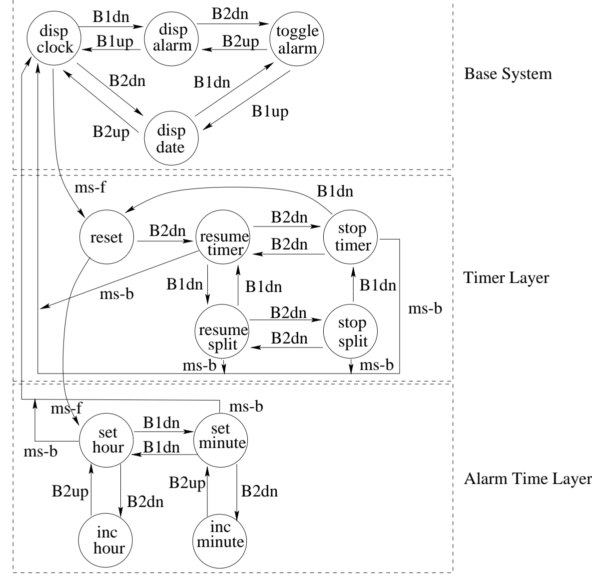


Figure 3: A collaborative design for a sportswatch.

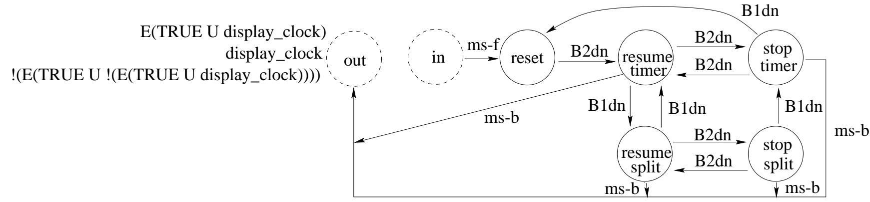


Figure 4: The timer extension with marking assumptions on the *out* state.

The classic CTL model checking algorithm [8] checks a property by marking each state with the subformulas of the property that are true in that state. After marking is complete, the formula is true of the design if its initial state is marked with the full property formula. If we can prove that an extension does not alter the markings of the base system states for a given property, then that property will hold in the composition of the base system with the extension as well. It suffices to show that the markings of the exit states in the base system interfaces are not altered, as all states which reach layer 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 *display_clock*), we use a model checker to extract the set of subformulas of the property that mark each state in the interface; these markings can be stored with the interface, and need not be re-computed on each extension. The following three formulas mark *dispclock*:

- E*(TRUE* U *display_clock)*
- *display_clock* (this implies the previous formula)
- *!(*E*(TRUE* U *!(*E*(TRUE* U *display_clock))))* (equivalent to AG(EF*display_clock*))).

We must prove that the extension will preserve the markings on the exit state from the base system. The CTL model checking algorithm marks states based on the markings of its successor states. As some extension states have transitions to the reentry state (in the base system), we need the reentry state's markings to compute the markings on the extension states. Our verification algorithm consists of assuming that the *out* state of the extension has the same markings as the reentry state, deriving the markings on the *in* state, and checking that those markings are the same as on the original reentry state; this approach is consistent with the standard backwards-reachability approach to model checking. We derive the markings on the *in* state by checking a property of the form AG$(in \rightarrow \phi)$ for each subformula $\phi$ of the property to be preserved. Figure 4 shows the sportswatch timer layer with the marking assumptions on *out*.

Model checking confirms that *in* retains the original markings of *dispclock*, so the property will be preserved upon composition.

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

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

- *!(*E*(TRUE* U *!(*E*(TRUE* U *display_clock))))*;
- E*(TRUE* U *display_clock)*;
- *display_clock*;
- *!(*E*(TRUE* U *!((starttimer* → E*(TRUE* U *stoptimer)))))*;
- *(starttimer* → E*(TRUE* U *stoptimer))*;
- E*(TRUE* U *stoptimer)*

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

### 3.4.1 Summary of Algorithm on Single State Machines

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

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

This algorithm, whose correctness proof we defer to a forthcoming technical report, was independently derived by Laster and Grumberg for reasoning about sequential decomposition of finite state machines [24]. Its correctness depends in part on all reachable states in the composed design lying in either the base system or the extension (an obvious point in the single-machine case, but one which becomes interesting in the multiple-machine case). For checking preservation of purely existential properties, this algorithm is unnecessary because sequential composition trivially preserves such properties (a simple observation, but one which Laster and Grumberg [24] did not note).

For our experiments, we simulated this algorithm using the VIS model checker. VIS does not support this algorithm directly, as there is no way to seed *out* with the assumed marking. Instead, we were forced to include a transition from *out* to the entire base system model; we did not include transitions from the base system to *in*. We verified that the markings on the actual reentry state (*dispclock*) did not change under this operation. As *in* was not attached to the base system, this approach is sufficient to argue that the verification would have gone through with the seeded markings (and no base system) had VIS supported that operation. Section 3.6 summarizes the issues that arose trying to use conventional model checkers for this sort of modular verification.

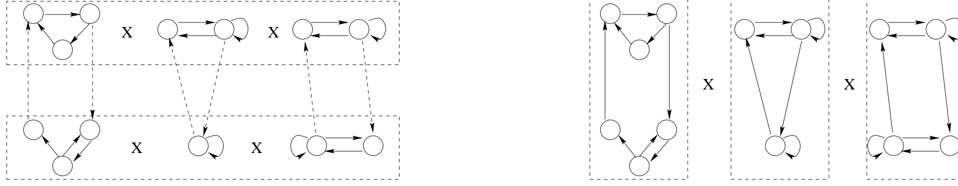

Figure 5: Two approaches to constructing composed systems.

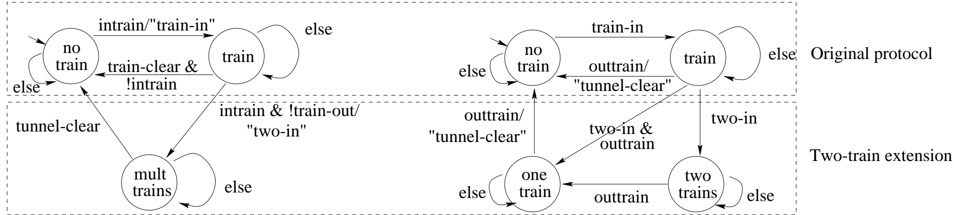


Figure 6: A collaborative design for a track-operator communication protocol.

## 3.5 Multiple-Machine Designs

The algorithm in Section 3.4.1, as well as prior research into verification under sequential composition, does not apply to FSATS because FSATS has multiple state machines in each layer. In practice, almost all interesting *collaborative* designs, by their very nature, will employ multiple state machines extensions per layer. When each layer contains a single state machine, extending a system with a layer corresponds to sequential composition of state machines. When layers contain multiple state machines, extending a system with a layer corresponds to a hybrid of sequential and parallel composition: the machines within a layer are composed in parallel (because they run together to implement a particular feature), but the layers themselves are composed in a quasi-sequential manner. The actual composition is not strictly sequential: this detail is at the crux of the verification problem for systems like FSATS.

Constructing a design by sequential composition is appealing because, as Section 3.4 shows, it supports independent verification of layers. Figure 5(left) shows a layered system constructed in this fashion. The construction provided in the formal model (the global composed state machine, Definition 4), however, is different. As Figure 5(right) illustrates, the construction first extends each base machine with its corresponding mixin, then composes the resulting machines in parallel. Clearly, we would prefer to compose systems according to the first construction because it supports layered 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 layered verification of designs with multiple state machines per layer. We must construct the parallel compositions representing each layer in such a way that composing them sequentially yields the state machine arising from Definition 4. This is possible only because most cross-product states in the composite system arise from cross-products within layers; this section notes the exceptions and how our methodology handles them.

This section motivates our algorithm for constructing parallel compositions within layers. Our algorithm is designed to create parallel compositions that can in turn be composed sequentially with other layers. We describe the algorithm by illustrating its behavior on a small example. We also evaluate this algorithm's ability to verify properties of layers in isolation. While many layers (including the FSATS layers) 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 system, rather than compositionally through the layers. We provide a characterization of these cases and a model-checking-based algorithm to determine whether properties can be verified compositionally. Section 3.5.1 presents our new example, which captures the salient characteristics of FSATS without necessitating as much explanation of the domain.

### 3.5.1 The Clayton Tunnel Protocol

We consider a layered design of a communications protocol between operators at either end of a train tunnel (see Figure 6). This protocol, taken from Holzmann's book [21], should already be familiar to those versed in the model

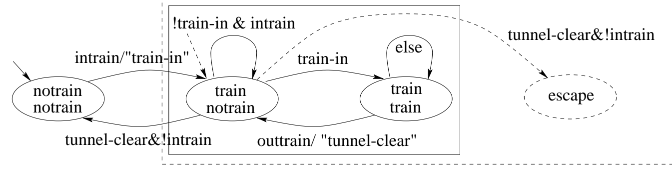


Figure 7: The cross-product state machine for the tunnel base layer. The exit subgraph for an interface containing both *train* states as exit states is enclosed in the solid box. The dashed box encloses the exit subgraph extended with an *escape* state for capturing the conditions under which control would leave the exit subgraph. The portion in the dashed box is part of the interface of the base system with both *train* states as exit states.

checking literature. 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 layer, 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 *train-clear* message to the inbound operator when a train exits the tunnel. The base layer 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 (in the same direction as the first train) before the first one 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 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 layer, we would like to prove that the inbound operator never livelocks thinking that there is a train in the tunnel ($\mathsf{AG}(\mathsf{EF}(inbb.state{=}notrain))$; this property requires all trains in the tunnel to eventually exit the tunnel, which we handle with a fairness constraint [9]. We can easily discharge this property of the base system; the challenge is to verify that the extension preserves it. For the extension, we wish to prove that once the inbound operator warns that there are two trains in the tunnel, it does not exit the extension until it receives a tunnel-clear message $(\mathsf{AG}\ ((inbmsg{=}two\text{-}in) \rightarrow \mathsf{A}(!(instate{=}out)\ \mathsf{U}\ (outbmsg{=}tunnel\text{-}clear))))$.

### 3.5.2 Composing the Extension in Parallel

The extension consists of the two state machines in the dashed box in Figure 6 (though with *in* and *out* states, as in Figure 4). We could form a naïve parallel composition of these two machines using a standard cross-product procedure [9]. This construction assumes that both machines start in their initial states (the *in* states) simultaneously. This assumption, however, is not necessarily valid. 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. Fortunately, we can derive this information 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 (which we call the *exit subgraph*, defined formally below) to drive transitions between the *in* states in the parallel composition. Figure 7 shows the exit subgraph for the

tunnel protocol. To verify preservation of properties under sequential composition, the exit subgraph includes a state that indicates when a transition would have left the exit subgraph; this state is labeled *escape* in Figure 7. While in practice this subgraph could be large, these graphs are small in FSATS (and presumably similar systems) because the actors decide to enter a particular extension at roughly the same time based on a tight sequence of message passing. Section 3.5.3 discusses a similar problem on the reentry states.

The following steps construct the exit subgraph:

1. Construct the cross-product of the base system machines.
2. Restrict the cross-product states and transition relation to those states that contain at least one exit state from some state machine in the cross product.
3. Add a new state *escape* to the resulting graph. From every state in the exit subgraph, add a transition to *escape* enabled on each condition that causes a transition outside of the exit subgraph. There are no transitions out of *escape*.
4. Identify all states (other than *escape*) with no incoming edges as initial states of the exit subgraph.

In the general case, the exit subgraph might not be connected. In designs such as FSATS, this subgraph is connected and has no transitions to *escape*. This is because the subgraph captures delays due to message passing before all actors enter an extension layer. In such cases, it can be used to sequentially compose a base system and an extension. Every state of the subgraph that contains exit state $exit_i$ enables transitions to $in_i$. Details appear in the full technical report.

Given the parallel composition of the extension machines constructed using the exit subgraph, we can attempt to verify the layer property using the 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. This happens because some history between the *in* states and the environment is lost. Specifically, the environment model must have the first train in the tunnel and the second train approaching the tunnel at the *in* states of the extension; the normal environment model starts with both trains approaching the tunnel. We can synchronize the environment model with the extension by composing the environment model with the base system before computing the exit subgraph. The initial states of the exit subgraph now contain states of the environment model; those states should be used as the initial states of the environment when verifying properties of the layer. 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 layered design. Layered designs offer the hope of testing layers in isolation. That testing, however, requires knowledge about the environment that will drive the layer. Our approach merely formalizes the problem of obtaining a restricted testing harness for layered 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.

### 3.5.3 Verifying Properties Compositionally

The preceding section identified two key issues in supporting verification of layers independently from their base systems: synchronizing initial states and restricting environment models. Applying both techniques allows us to verify that an extension satisfies a given property relative to an interface to a base system. This does not address our entire problem, however, as we still must characterize when the properties of the composition of this layer with a base system can be verified via sequential composition, rather than on the global composed state machine.

The algorithm for checking property preservation under sequential composition requires that all states that are reachable in the composed system are contained in one of the layers being composed. For compositional verification to work, we must characterize when the reachable states of the composed system are contained in the reachable states of the union of the base system and the extension. In the general case, the desired result seems unlikely. Just as the actors do not enter an extension simultaneously, they do not exit from the layer simultaneously. The asynchronous exits may create reachable states in the composed system that are not contained in the extension. Worse still, these states may lead to states that become reachable in the base system only after composition. Either case would break our proposed layered verification methodology.

Fortunately, the collaborative 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. Thus, with appropriate modeling of the reentry states in the extension, the sequential composition of the base system and the extension could capture the full global state space, as required for layered verification. We capture this model with a reentry subgraph that is computed in similar fashion to the exit subgraph; transitions to reentry states enable transitions in the reentry subgraph. Using the exit and reentry subgraphs, we can offer a CTL characterization of the cases in which our methodology is insufficient. When our methodology does not suffice, we will have to check the properties in the full composed system.[1]

The constraints that indicate when a property preservation can be confirmed (using a similar strategy as in the single state machine case) in layered fashion are as follows:

1. The *escape* state in the exit subgraph is not reachable under the restricted environment model.
2. The *reentry* states eventually synchronize: that is, once one layer machine reaches its reentry state, it remains there until all layer machines have reached their reentry states. This constraint is easily expressed as a series of CTL formulas to check of the model, one for each state machine in the extension:
$$\mathsf{AG}\ (reentry_i \rightarrow \mathsf{A}[reentry_i\ \mathsf{U}\ reentry_1 \wedge \ldots \wedge reentry_k])$$

We omit the proof that these conditions are sufficient to prove a correspondence between the two constructions of the global composed state space due to space constraints; the technical report will contain the full details. Intuitively, the proof consists of an argument that, under the above constraints, all reachable states under the first construction are reachable states in either the extension or the base system under the second construction. The interesting cases of this proof involve global states with some components in the base system and some in the extension. The conditions listed above restrict all such states to lie in the extension including the exit subgraph.

### 3.6 Implementation

We have conducted all the model checking tasks described in this paper. For this, we used the symbolic model checker VIS [32]. 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 layers. For the paper's examples, the time and space usage are negligible.

Section 3 describes how we simulated the modular verification scenario while in fact attaching extensions to, potentially, the entire base system. This is because existing model checkers do not appear to be designed for extension to verifying open systems. For instance, they do not provide a way to query and assert properties on specific states. Expressing our extension layers 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. Finally, building the exit and reentry subgraphs was difficult in VIS's symbolic framework. 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 layered design languages could work around the limitations of Verilog, but the limitations of the symbolic framework are harder to surmount.

Some properties can be verified in layers without the full power of model checking. For instance, simple properties that ensure a system 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.

## 4 Conclusion and Future Work

Layered designs, which concentrate on the step-wise refinement of a system's features, offer an alternative to traditional object-based designs. They have arisen in several contexts, methodologies and applications, and appear to

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

be especially promising in the context of software product-lines. Layered approaches share many of the software construction advantages of more traditional components.

This paper has explored how layered software designs require a different form of modular verification. We demonstrated that object-based decompositions of systems into modules that are concurrently or sequentially composed are inappropriate for layered designs. We also showed that layered, feature-based designs are actually quasi-sequential compositions of parallel compositions, and explained how certain constraints can make their verification tractable. We believe these constraints are reasonable because many applications appear to satisfy them. The resulting verification methodology minimizes the work expended to verify compositions relative to the work done verifying individual layers.

We have concentrated solely on model checking because we want to understand the strengths and limitations of algorithmic verification on layered designs. Our experience suggests that extant model checkers have not been designed to be extended for such tasks. (Certainly, a custom model checker is necessary to complete the verification of the entire suite of FSATS layers.) A related question is how to extend our approach to handle LTL formulas; for technical reasons, we have only considered CTL properties.

Layered designs can benefit from a broader scope of verification techniques. Early work on dependencies between layers [3] must be formalized and incorporated into any validation framework. We have encountered some layered designs involving complex data invariants that will likely be more amenable to theorem proving. While model checking captures and can verify the salient properties of the FSATS suite, it can be overkill. Preserving certain properties requires only simple results such as freedom from livelock or non-modification of particular variables. In these cases, simpler tools such as reachability engines and type systems may suffice. We expect further work with a richer set of designs to help us identify when the full power of our current methodology is required.

# References


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

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

[4] Batory, D., C. Johnson, B. MacDonald and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, June 2000.

[5] Batory, D. 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, October 1992.

[6] Biagioni, E., 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] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992.

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

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

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

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

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

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

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

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

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

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

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

[19] Grumberg, O. 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.

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

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

[22] Kiczales, G., 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.

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

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

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

[26] Mezini, M. and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 97–116, October 1998.

[27] Pasareanu, C. S., 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.

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

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

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

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

[32] The VIS Group. VIS: A system for verification and synthesis. In Alur, R. 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.

[33] van Renesse, R., 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.

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