The Influence of Software Module Systems on Modular Verification

Harry Li[2], Kathi Fisler[1] and Shriram Krishnamurthi[2]

[1] Department of Computer Science, Worcester Polytechnic Institute
[2] Computer Science Department, Brown University

Abstract. The effectiveness of modular model checking for hardware makes it tempting to apply these techniques to software. Existing modular techniques have been driven by the parallel-composition semantics of hardware. New architectures for software, however, combine sequential and parallel composition. These new, *feature-oriented*, architectures mandate developing new methodologies. They repay the effort by yielding better modular verification techniques.
This paper demonstrates the impact of feature-oriented architectures on modular model checking. We have implemented a model checker and applied it to a real software system to validate our prior, theoretical work on feature-oriented verification. Our study highlights three results. First, it confirms that the state-space overhead arising from our methodology is minimal. Second, it demonstrates that feature-oriented architectures reduce the need for the property decompositions that often plague modular verification. Third, it reveals that, independent of our methodology, feature-oriented designs inherently control state-space explosion.

1 Introduction

Recent advances and successes in the computer-aided verification of hardware fuel the desire to effectively apply these ideas to software. The goal is to develop models and analyses that simplify early detection of software design errors without disrupting the design flow. Early detection requires that techniques for *verifying* software be closely intertwined with the techniques and tools for *designing* and *producing* software. Verification techniques and development techniques must therefore evolve together if verification is to be viable for substantial software systems.

The hardware model-checking community has long demonstrated that the bond between design and verification can go beyond necessity to symbiosis: in particular, that decomposing designs according to their modular structure can reduce an intractable verification problem into a collection of tractable ones. The results of the tractable verifications can be combined into results on the otherwise-intractable overall design. The general idea of modular verification applies to software as well, but with a technical twist: *modules in software design are evolving towards a model that violates the assumptions underlying existing modular verification techniques.*

Traditional modules encapsulate participants (or *actors*) and contain the code that the actor needs to implement the features (operations/services) of the system. Modern software modules encapsulate *features* rather than actors. These *feature-oriented designs* realign the module boundaries so that all of the code pertaining to a single

operation lies in the same module; the modules therefore *cross-cut* actors. Researchers have proposed feature-oriented modules under many names (*refinements* [6], *units* [15], *aspects* [23], *collaborations* [27], *hyper-slices* [28], and others); some have spoken of feature-oriented designs (or *feature engineering*) in more general terms [32]. Ongoing research on feature-oriented modules shows that they simplify key software engineering problems such as configurability, maintainability, and evolution [3, 15].

Since features often operate exclusively from one another, feature-oriented modules do not compose in parallel. Instead, their composition model employs a certain combination of parallel and sequential composition. Existing modular verification techniques assume either purely parallel or purely sequential composition; accordingly, none of them apply to feature-oriented designs. In previous work, we proposed a methodology for modular verification of feature-oriented designs [17]. The existence of this methodology, however, does not address the more crucial practical question: do feature-oriented modules simplify or facilitate verification in practice?

This paper argues that feature-oriented modules are better suited for modular verification than traditional module systems. We present a case study on verifying a substantial feature-oriented software design with our new modular verification methodology. We base our claims about the superiority of feature-oriented modules for verification on the following claims:

- They simplify the problem of decomposition in verification because such modules naturally align with properties. This reduces, and often even eliminates, the current need for property decomposition in modular verification.
- They provide a felicitous framework for composing results of modular verifications into results on whole systems, while avoiding some of the circularity difficulties inherent in classical modular verification work.
- Their design discipline appears to even inherently control state-space explosion.

Section 2 motivates and illustrates feature-oriented modules by describing the software system that we use in the case study. Section 3 summarizes our methodology for feature-oriented modular verification. Section 4 presents our case study using this methodology to verify the design described in Section 2. Section 5 discusses related work, and Section 6 offers concluding remarks and outlines future work.

2 FSATS: An Example of Feature-Oriented Design

FSATS is a simulator for command-and-control missions. Missions involve a hierarchy of (military) personnel; each person in the hierarchy commands a set of weapons. In a simulated mission, certain personnel identify potential targets and initiate a communication protocol to determine who (if anyone) will attack the target. This decision is based on a series of factors including the nature and location of the target, as well as the availability of weapons at each point of the hierarchy. Once a person accepts responsibility for a mission, he commands his weapons to attack the target.

One of the main challenges a programmer would experience in implementing FSATS is that the personnel and weapons hierarchies need to be sufficiently flexible to simulate a variety of military scenarios. This requires several kinds of customizations:

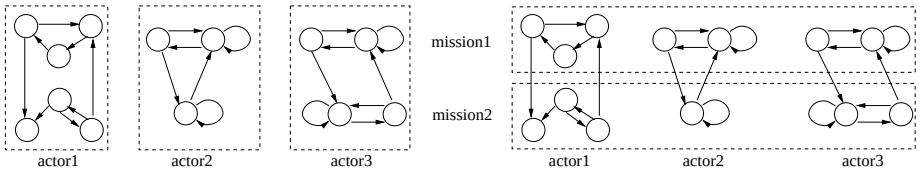

Fig. 1. Two modularizations of FSATS: actor-oriented (left) and feature-oriented (right); the dashed boxes delimit module boundaries in each figure.

- Certain terrains preclude certain classes of weapons; the weapons controlled by each person must change according to the terrain under simulation.
- Different branches of the military employ different personnel hierarchies; each person's superiors in the hierarchy must therefore be flexible.
- Different situations may require personnel to respond differently to the same nature and location of target; thus the algorithm for deciding whether someone can accept a mission requires flexibility, sometimes on-the-fly.

Constructing separate simulators from scratch for each potential scenario is infeasible. FSATS implementations therefore need to be customizable along all of these lines with minimal reconfiguration effort. Recompilation is acceptable when building a new simulator, but modification to existing code is not. Batory, Johnson, MacDonald, and von Heeder [4] designed and implemented FSATS using feature-oriented modules to endow it with these capabilities. This implementation used Batory's JTS system [5], a Java front-end developed to support feature-oriented modules. This section uses their decisions and observations to motivate (Section 2.1) and define (Section 2.2) feature-oriented design.

2.1 Feature-Oriented Designs

FSATS consists of personnel and weapons (collectively called the *actors*) and missions for firing on targets (the *features* or operations that the actors cooperate to implement). For each actor/mission pair such that the actor participates in the mission, FSATS contains some code fragment(s) implementing the actor's role in the mission. The architecture organizes these code fragments into cohesive constructs, such as classes and modules. Viewing the actor/mission pairs as a grid, two organizations jump to mind (Figure 1): modules can align with actors/columns (actor-oriented modules), or modules can align with missions/rows (feature-oriented modules). The figure shows the code fragments as state machines, which is how FSATS expresses its mission protocols.

To motivate the appeal of feature-oriented modules, consider the problem of adding or removing missions from a simulator. For a given set of target conditions, several actors are involved in deciding which mission to execute. Altering missions under actor-oriented modules therefore requires modifying the modules for each actor involved in the mission. As the code corresponding to a single mission may not be cleanly isolated

in the original code (since multiple missions may involve similar decision-making processes), this editing operation is potentially expensive. With feature-oriented modules, in contrast, each module encapsulates code for a mission centered around a particular weapon under a certain set of conditions. To remove a weapon from the system, a programmer can simply re-compose the system without the missions that use a weapon; the original implementor performed the necessary decomposition, so no editing of code is required.

Feature-oriented modules have been called *collaborations*, since a module encapsulates the code through which the actors collaborate to perform an operation. We adopt the term *collaboration* in the rest of this paper. In FSATS, each actor/mission code fragment is a class. A collaboration is therefore an ordered tuple of classes, one per actor. Collaboration composition connects the classes for each actor via object-oriented inheritance. The resulting (single) class contains all of the code needed to implement each mission for that single actor.

FSATS's requirement of flexible personnel hierarchies mandates that classes within collaborations have parameterized super-classes. Assume that battalion leaders report to brigade leaders in one simulator and to division command in another. These simulators require different collaborations for their core communications protocols. A designer implementing a mission involving battalions does not know which communication collaboration to use; that decision happens at system-composition time.[1] The designer therefore cannot fix the super-classes of the classes in his collaboration; he can, however, impose constraints on them through interfaces. Classes with parameterized super-classes are called *mixins* [8, 19, 30, 34].

Collaborations comprised of mixins provide the flexibility needed to implement FSATS. Different FSATS simulators are built by selecting weapons and communications collaborations and composing them to form a complete simulator. As described here, collaborations obey the characteristics of components [19], such as separate compilation, multiple instantiability and external linkage. A brief sampling of other successful designs in this domain includes protocol layers and database modules [6, 7, 33], a programming environment [14], test-bench generators [21] and verification tools [18, 31]. The growing application of collaboration-based architectures also reflects in the increased language support for programming with collaborations.

2.2 A Formal Model of FSATS

Having motivated the overall architecture of FSATS, we now describe a more formal model of collaborations, their interfaces, and their compositions that we use in our verification methodology. In FSATS, two pieces of code implement a particular actor's role in a mission. The first is a state machine fragment that specifies a mission-specific communication protocol. The second is a set of rules that govern whether an actor is equipped to accept a particular mission (based on his weapons' status and capacity). Our case study verifies properties of the communication protocol, not of the weapons selection rules. We therefore adopt a simpler view of FSATS in which each collaboration

[1] In other words, collaborations are composed through *client-controlled* or *third-party* linking.

consists of a tuple of state machine fragments and an interface for composing collaborations; each state machine fragment extends an existing (base) state machine by adding nodes, edges, and/or paths between states in the base machine.

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

The following formal definition from our earlier paper [17] makes our model of collaboration-based designs precise. The definitions match the intuition in the figures, so a casual reader may wish to skip the formal definition.

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

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

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

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

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

Given a base system B, one of its interfaces I, and an extension E, we can form a new system by connecting the machines in E to those in B through the states in I, as shown in Figure 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 states in the constituent machines of base systems and extensions are distinct.

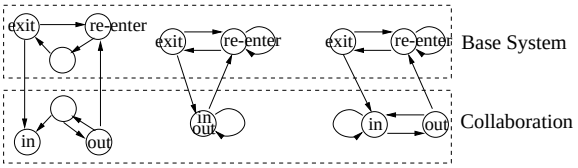

Fig. 2. Collaborations, interfaces, and composition

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

Modular verification succeeds when the designer can isolate a portion of a design that is relevant to a property. Assume all of the actors in a system participate in executing a feature that we wish to verify. If the design uses an actor-oriented architecture, the modular structure naturally decomposes the design into individual actors. But the property is typically in terms of the *entire feature*, not the individual actors. Thus, the verification engineer must *decompose the property to align with the modular structure.* Experience shows that this task can be extremely difficult in practice because it is hard to isolate how one particular actor (or small set of actors) contributes to satisfying a property. Furthermore, actor-based property decompositions can induce circularity in the assumptions of behavior between modules [10].

In contrast, collaboration-based designs often avoid both of these problems because the modules naturally decompose around features. If the property concerns a feature, collaborations isolate the relevant portion of the system by design. Ideally, we should be able to verify a property of a feature by analyzing the collaboration that implements that feature in isolation from the rest of the system. Our methodology provides a mechanism for doing this.

What about properties that concern actors rather than features? Wouldn't an actor-oriented architecture be more suitable for proving those properties? Collaborations actually support both actor-oriented and feature-oriented decompositions. A full design

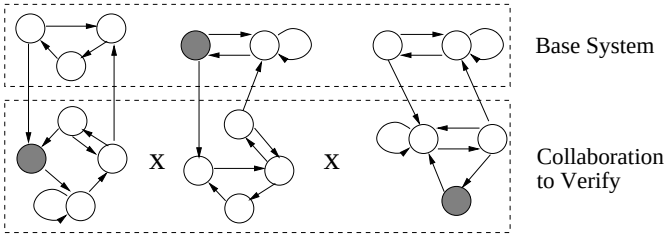

Fig. 3. Constructing collaboration cross-products to enable model checking. The shaded states represent a reachable state in the overall system; this cross-collaboration state arises during the transition from one feature to another.

is composed from a set of collaborations, which are tuples of mixins. If a property concerns the behavior of a single actor across multiple features, a verification tool can extract the actor's mixins from the collaborations, compose them via inheritance, and verify the property against the result. In short, collaborations can be composed either vertically or horizontally (as shown in Figure 1) as needed. Designing systems to support both actor- and feature-oriented composition does, however, force designers to break actors down into feature-sized pieces. The variety of systems that designers have built using collaborations suggests that programmers are willing to do this work in exchange for the benefits associated with collaboration-based designs.

A Methodology For Verifying Collaborations Modularly

This section describes our methodology for verifying properties against individual collaborations using CTL model checking. The methodology currently supports the following activities:

1. Proving a CTL property of individual or compositions of collaborations.
2. Deriving *preservation constraints* on the interface states of a collaboration that are sufficient to preserve each property after composition.
3. Proving that a collaboration satisfies the preservation constraints of another collaboration (or existing system). We establish preservation by analyzing only the extension, not the composition of the two collaborations.

The main challenge in the methodology lies in the first activity. In order to model check a property against a collaboration, we need a single state machine for the global cross-product of the machine fragments in that collaboration. We discuss the issues in constructing this cross product below. The second activity involves recording some information during the CTL model checking process. The third involves mostly routine CTL model checking, with an initial seeding of labels on certain states of a design. We use CTL rather than LTL because the CTL semantics supports the state labelings that we need for our methodology; adapting our methodology to LTL is an open problem.

Let us examine the task of constructing the cross-product of the state machine fragments in a collaboration. Figure 3 illustrates the situation: we wish to verify the lower collaboration in isolation from the upper one. Since actors operate in parallel within a collaboration, we must therefore construct the cross product of the state machine fragments in the lower collaboration (as the "x"'s between the fragments in the lower collaboration indicate). Cross-product constructions begin with a set of initial (cross-product) states. What, though, are the initial states of the lower collaboration? Only the base system contains the initial states for the completed design. For other collaborations, only their "in" states (in the interface) give any indication of how to start running the collaboration.

It is tempting to assume that all actors will enter the collaboration for a feature at the same time: that is, to assume that the tuple of "in" states from the interface is reachable. Unfortunately, practice violates this simplistic assumption. In FSATS, for example, the person accepting a mission enters the collaboration for that mission and sends a message to that effect along the chain of command. As other people receive the message, they too enter the collaboration. While it is true that once one actor enters the collaboration the others will (eventually) follow, they do not enter the collaboration all at once. Figure 3 illustrates this situation through the shaded states; two actors have entered the collaboration, while the middle actor has not yet made that transition. Detecting the initial states of the collaboration to verify is therefore non-trivial. The shaded states also illustrate that some reachable cross-product states span collaborations. Such states are reachable only during the transition from one collaboration (feature) to another.

In FSATS, all mission collaborations (all collaborations other than the base system) attach to the base system. We have presented a formal algorithm that exploits this organization to identify the cross-collaboration states; our methodology uses these states to drive the cross-product construction for the collaboration [17]. The construction includes the cross-collaboration states with the collaboration cross-product, which guarantees that our methodology visits all reachable cross-product states; details appear in our earlier work [17]. This process may add a few states to the state machine fragments in each collaboration; we explore the impact of these extra states experimentally in Section 4. Once we identify the possible initial states, we use a standard cross-product construction to obtain a single state machine suitable for model checking the collaboration.

Having computed the cross-product for a collaboration, we use the standard CTL model checking algorithm [9] to verify properties. Proving that composition preserves the property is the next challenge. This is where collaboration-based verification diverges from standard approaches to modular verification. Under parallel composition, modular verification techniques assume that composition does not add new behaviors to a module. This is a reasonable assumption since the states of two modules interact only through a cross-product construction. In contrast, composing collaborations adds transitions, and thus behaviors, to states in a given module. These extensions are a natural and important part of collaborative designs. This characteristic, however, inhibits the use of modular verification techniques based on parallel composition.

Fortunately, the limited communication between collaborations—which occurs only at the interface states between the collaborations—reduces modular verification to a

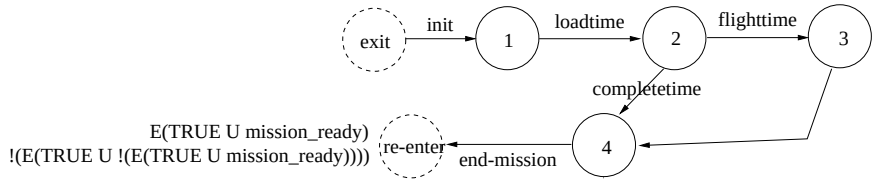

Fig. 4. A example of the methodology. The dashed states are placeholders for the interface states to which the collaboration shown attaches. The formulas next to the "re-enter" state are the seeded labels. CTL model checking determines labels on the "exit" state based on the seeded labels.

form of sequential verification. We use Laster and Grumberg's algorithm [26] for compositional model checking under sequential composition for this step. Briefly, when model checking a property against a collaboration, we record the labels that the CTL model checking algorithm assigns to the interface states. When we attach a new collaboration to those states, we check that the new collaboration will not invalidate any of those labels. We perform this step by attaching two dummy states to the new collaboration (one each for exit and re-entry), seeding the dummy re-entry state with the saved interface labels, and using the CTL model checking algorithm to derive labels on the dummy exit state (see Figure 4). If the derived labels are consistent with the recorded labels, the composition will preserve the property of the original collaboration.

With the exception of seeding states with properties, our methodology uses standard CTL model checking algorithms. The contribution of our methodology lies in techniques for computing cross-products of collaborations and in identifying necessary constraints on collaboration interactions to guarantee that our approach is sound with respect to a conventional actor-oriented modularization. We base soundness on the claim that our method would explore the same set of global states as in an actor-oriented, parallel composition of the state-machine fragments. In other words, the state spaces obtained by the two modularizations shown in Figure 1 are equivalent. Our prior work [17] presents the additional constraints needed to achieve soundness; intuitively, these constraints require forms of synchronization between actors at feature boundaries.[2]

4 Results on FSATS

Our FSATS case study was designed with several goals in mind:

- To validate our modular verification methodology on a significant software example. FSATS suits this role well: a full FSATS system contains at least 14 actors participating in at least 15 different mission types. The case study reported in this paper used 3 representative mission types over 14 actors.
- To determine the levels of state-space reduction we can achieve through feature-oriented modular decomposition.

[2] These constraints are, incidentally, also necessary for modular testing.

– To determine the overhead due to our verification methodology.
– To explore whether feature-oriented modules provide decompositions that naturally align with properties.

This case study employed a base system containing the core communications protocol and three missions: one in which the battalion fires a mortar, one in which a platoon attacks with an artillery unit, and one in which the division commander fires a set of rocket launchers. The mortar and artillery missions embody simple protocols and yield small state machines. The rocket launcher protocol is more complicated because launchers must scurry out of hiding places in order to fire, then return to hiding places to reload during an attack. The coordination of launchers across hiding places gives rise to a protocol similar to *cache coherence*: the division officer must know where the rockets are at all times, and no two rockets can hide in the same spot at the same time.

We chose these three layers for several reasons. First, the mortar and artillery layers share some common design variables, so there is potential for property clashes when these modules are composed. Second, only a portion of the command hierarchy participates in deciding whether to use mortar or artillery, so we have the potential to eliminate unnecessary participants, as we would do in a standard parallel decomposition. Finally, the rocket launcher collaboration is substantially larger than the other two; ignoring this collaboration when reasoning about either of the other two collaborations should noticeably moderate the resources required during verification.

4.1 A Model Checker Supporting the Methodology

Although our methodology centers around the standard CTL model checking algorithm, existing CTL model checkers do not support it well. Existing checkers embody a closed-world assumption, in which all variables involved in the model are generated within the model. This assumption is invalid in collaborative modular verification. When we verify that composition does not invalidate existing properties of collaborations, we must seed states with non-trivial CTL formulas that would be true in that state after composition. Existing model checkers do not permit this seeding, though; they instead require all formula labels to be derived during model checking. We could augment the model to accomplish seeding—by adding an automaton sufficient to generate the desired labels—but this change is both drastic and painstaking to perform manually; it also artificially increases the size of the model.

We have implemented a prototype custom model checker that allows seeding of states with arbitrary CTL formulas. If no states are seeded, the model checker behaves as a conventional CTL checker. If states are seeded, then model checking results are valid under the assumption that the seeded formulas hold in their corresponding states. The rest of the methodology discharges this assumption.

Our checker also confirms that models satisfy the constraints that our methodology requires for soundness. These constraints involve checking reachability of certain specific states that we identify based on the interfaces between collaborations. It also confirms that collaborations do not deadlock; this is important for the correctness of our methodology (otherwise the sequential composition does not result in a continuously running system). We have written the prototype in PLT Scheme [14].

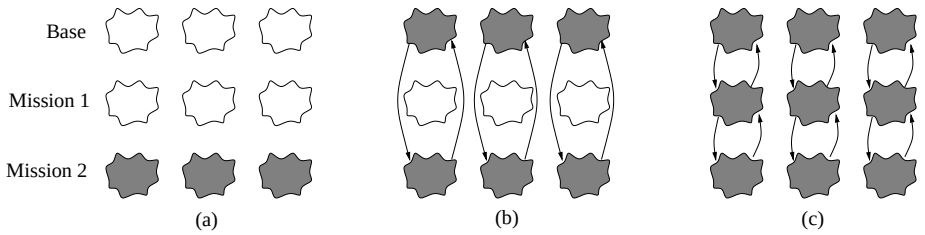

Fig. 5. State-spaces contrasted in our experiments. Each shape represents a state-machine fragment. The shaded shapes indicate which fragments are included in each comparison.

4.2 Experiments on the Impact on State-Space Size

Our first experiment assumes that a feature is primarily characteristic of one collaboration. We would want to verify that property against that collaboration alone for two reasons. First, until we have established its correct implementation, it isn't worth verifying against other collaborations. Second, as independent developers, we may not even know the other collaborations at development time; only the final system integrator will know all the collaborations.

Figures 5-a and 5-b iconographically depict two approaches to constructing a state space for this verification. The collaborative design of 5-a allows us to consider just the collaboration of interest. The system in 5-b results from cross-producting actor-oriented machines. For our experiment, we obtained these latter machines by manually linking the machine fragments across collaborations, as we discuss in Section 3. The table below presents this comparison.[3]

Mission	States in Collaboration Cross-Product (Fig. 5-a)	States in Collab+Base Cross-Product (Fig. 5-b)
Mortar	23	127
Artillery	29	124
Rockets	4,888	10,032

A realistic system consists of numerous missions, not just one. Therefore, a more thorough assessment of state spaces would study the machine sizes that result from composing multiple missions. We contrast two verification tasks. The first verifies properties against the cross-product of an actor-oriented decomposition. This may be given by the programmer; in FSATS, we construct the actors by combining their machine fragments from multiple collaborations (Figure 5-c). The second verifies properties against each of the collaborations separately, in the manner described in section 3. The first two columns of the following table present information on the actor-oriented systems; the third column contrasts this against the sum of the sizes of each collaboration cross-product (because the verifications are performed independently). We use an asterisk to indicate that a computation did not complete in the available memory; the number

[3] These sizes do not include the environment models that may be needed for model checking.

of states reported is a lower bound on the total number of states, but the number of transitions may include multi-edges (unlike the transition data in the completed runs).

Missions (Plus Base)	States in Whole System (Fig. 5-c)	Transitions	(Sum of) States in Individual Collabs
Mortar, Artillery	237	339	23 + 29
Mortar, Rockets	114,300*	1,132,069*	23 + 4888
Mortar, Artillery, Rockets	160,472	314,694	23 + 29 + 4888

These data underscore that verifying properties of multiple missions results in additive state-space growth with collaboration-based verification; the growth with no modular verification is potentially multiplicative.

The results from these two tables clearly establish that feature-oriented decomposition can result in substantial state-space reductions. As we would expect, the savings grow more impressive as we add missions to the simulator, because new missions do not affect the state spaces of individual collaborations. Decomposition around collaborations therefore controls the growth of state spaces in model checking.

The data in the first table indicate that our methodology is indeed effective in restricting the number of states from the base system that need to be visited while model checking a collaboration. The following table contrasts the total size of the base system (restricted to the actors involved in the mission) with the number of base system states needed to drive the construction of the collaboration cross-product:

Mission	States in Base Cross-Product	States from Base To Drive Collaboration
Mortar	12	1
Artillery	73	4
Rockets	63	4

In the course of our experiments, we discovered that the choice of interface states can affect our methodology's overhead. In our first model, the Mortar weapon had an almost trivial state machine in the base layer: one state for starting a mission and another for ending a mission. These two states were the interface states to which we attached collaborations involving mortars. The mission starting state was also the initial state for the mortar in the base system. Having the same state be both the initial state of an actor (in the base) and the interface "exit" state caused the number of overhead states to bloat artificially to include the entire base system cross product; this had 12 states in the case of the Mortar mission actors. After introducing a separate state to use as the interface state, the overhead dropped to 1 state, which was what we expected it to be.

Our experiments yielded another surprising result. Based on the sizes of the state machines for the individual actors (between 5 and 100 states per mission for each of 14 actors), we expected the number of cross-product states in each simulator to grow dramatically as we added missions. While we did observe noticeable growth, particularly after adding the rocket mission, the growth was not strictly multiplicative. We later realized that *the synchronization between actors needed to properly implement collaborations naturally limits state explosion* (since the requirements limit the number of

global states involving states from multiple collaborations). This characteristic of collaborations is orthogonal to our methodology, and instead reflects a general benefit of feature-oriented architectures.

To validate this claim, we removed some of the synchronization between actors at collaboration boundaries and recomposed the simulators. Removing the synchronization allows one actor to start a new mission upon completion of an old one, without negotiating with the other actors. This led to a noticeable increase in state-space size. For example, the unsynchronized base+mortar simulator had 257 states (versus 127), while the unsynchronized base+rocket simulator had 63,657 states (versus 10,032). We were unable to finish computing the unsynchronized size for the simulator with all three missions within our available memory.

4.3 Modular Verification Experience

Applying existing modular verification techniques can be difficult in practice due to the need to decompose properties while avoiding circular arguments. We therefore wanted to gauge whether our methodology helped or hindered the verification process. This section discusses our observations from using our model checker to verify several properties of FSATS. We do not discuss the actual properties or running times and memory usage. The properties are standard CTL invariants and eventualities. We omit the resource usage data for two reasons. First, we are using explicit-state model checking, so state-space size is a reasonable predictor of performance (unlike with BDDs). Second, our current tool is a proof-of-concept prototype for our algorithms so we expect the resource usage would be artificially high.

Assume a user wants to verify a property about the mortar mission, such as "once started, the mortar eventually fires all rounds". The user provides our model checker with three pieces of information: the base machines for the FSATS actors, the collaboration implementing the mission, and the property to verify. Our tool automatically constructs the cross-product of the collaboration (using information from the base machines), checks whether the collaboration satisfies needed synchronization restrictions, and calls the model checker. Error traces, if any, are of the usual flavor and are expressed relative to the collaboration cross-product. The property and the constraints needed to confirm that it holds after composition are stored with the collaboration. When a user composes a new collaboration onto a system, our tool automatically confirms that existing properties of the collaboration and system are preserved.

In short, a user's interaction with this system is similar to that with a conventional model checker, with the exception of indicating which collaboration each property should be proven against. There is no need to decompose the system or the property. The tool manages all of the assumptions required for modular verification (this entails deriving labels during earlier model checking runs). There is no danger of introducing circularity, because the user does not need to introduce any information; all of the decomposition information comes directly from the design architecture.

Our tool's automated checks for synchronization requirements (mostly various reachability checks) also helped us detect some design errors. For example, the re-entry interface states must be reachable within a collaboration. Our early model of FSATS had some errors that violated this restriction. None of the properties we tried to check would

have detected the problem. Thus, our methodology does provide some simply sanity checks on designs that can help locate real errors in system models.

5 Related Work

Several verification techniques use design information to restrict the state space to the portion relevant to a given property. *Cone-of-influence reductions* [25] use dependence analyses between variables to eliminate portions of the state space. These analyses retain portions of the state space needed to reach the relevant portion of the design from the initial states; our method eliminates most of the states traversed from the initial states to the point of entry to a collaboration. Cone-of-influence reductions are also less effective if multiple parts of the design involve the same variables. Multiple FSATS layers refer to shared variables that also occur in the properties we wish to verify; all of these layers would be explored under a cone-of-influence reduction.

Variants of code layering have been used in both software engineering and verification contexts. The term "layered architecture", however, generally assumes that each layer refines a more abstract layer already in the system. Such assumptions correspond to abstraction or refinement layers in verification, in which one layer is shown to subsume the behavior of another [25]. This work is orthogonal to ours, which does not require any abstraction relationship between collaborations.

Several researchers have described modular verification techniques based on parallel composition [16, 20, 24, 29]. Some preliminary research [2, 11, 26] considers modular model checking under sequential composition, which is closer to the model used in software. Laster and Grumberg's approach [26] handles designs with only one state machine; it also lacks a design framework, such as collaboration-based design, to drive the decomposition of the design. Subsequent work obtains this decomposition from hierarchical state machines [2] or StateCharts [11], but still considers designs with only one state machine. Our work, in contrast, includes *multiple state machines per collaboration*, which greatly complicates the verification problem. Alur and Yannakakis cite the problem of sequential verification over multiple state machines as open for future work [2]. Alur *et al.* [1] discuss analysis techniques for sequential refinements within modules that are composed in parallel; their work, unlike ours, does not support *coordination* between sequential refinements across modules. None of these works compares the state space sizes in their techniques against those of traditional model checking.

Work on pre- and post-condition verification in theorem proving is another form of modular reasoning under sequential composition. Such work views code at the level of individual, stand-alone functions and instructions, rather than at the level of coordination between multiple actors in a system.

6 Conclusions and Future Work

Modular verification is an attractive approach to managing state explosion. Modules, by design, delineate somewhat independent portions of a system. In theory, we should be able to exploit this independence to decompose intractable verification problems into tractable ones about each module (or small groups of modules). Experience shows that

modular verification is extremely difficult to use in practice. The main challenge lies in *property decomposition*: the need to decompose a property of a system into sub-properties of its modules. Traditional modules reflect *physical* independence (different devices on a chip, for instance) rather than *behavioral* independence. Since properties concern behavior, conventional modular structures are misaligned with properties and make modular decomposition difficult.

Collaborations are modules that encapsulate code involved in the same operation in a system. They have received increasing attention in software engineering because their separation of behavior simplifies software evolution, configuration, and maintenance. This paper explores the effect of these designs on modular model checking, especially on state space sizes and on the need for property decomposition.

We present a case study of applying a new model checker we have developed to a real command-and-control simulator called FSATS. The results are extremely positive. Collaborations dramatically reduced the size of the state space to be explored during model checking, while requiring no property decomposition. Furthermore, we observed that the programming discipline of collaborations requires certain synchronizations that naturally control state-space growth. The ease of modular verification in this framework, combined with the measured reductions in state space sizes, suggest that collaborations provide more useful modularizations for verification than conventional modules do.

We plan to continue our study along several lines. First, we need to confirm our hypothesis that collaboration-based decomposition achieves greater reductions than cone-of-influence analysis. Second, we need to extend our current methodology to support data-intensive designs, instead of just control-intensive ones. FSATS will continue to be an interesting example for this effort, as it involves a combination of a control-intensive communications protocol between personnel and data-intensive decisions within the protocol. Third, our current methodology assumes that collaborations specify control-flow via state machines. We would like to extend existing work on deriving state machines from source code [12, 13, 22] to extract collaboration-oriented models. A related question asks whether our collaboration-based organization is too restrictive; it may be possible to extract collaboration-like behavior from code that is composed in parallel, perhaps by examining synchronization points.

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

5. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *International Conference on Software Reuse*, June 1998.
6. 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.
7. 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.
8. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, Mar. 1992.
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. 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.
13. 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.
14. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001. To appear.
15. 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.
16. 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.
17. K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Symposium on the Foundations of Software Engineering*, Sept. 2001.
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, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.
20. O. Grumberg and D. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
21. Y. Hollander, M. Morley, and A. Noy. The *e* language: A fresh separation of concerns. In *Proceedings of TOOLS Europe*, Mar. 2001.
22. 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.
23. 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.

24. 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.
25. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
26. K. Laster and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1998.
27. K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Mar. 1999.
28. H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM, Apr. 1999.
29. 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.
30. Y. Smaragdakis and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.
31. K. Stirewalt and L. Dillon. A component-based approach to building formal-analysis tools. In *International Conference on Software Engineering*, 2001.
32. C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, Dec. 1999.
33. 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.
34. 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.