Verifying Cross-Cutting Features as Open Systems*

Harry Li†
Computer Science Department
Brown University
Providence, RI, 02912 USA
hcli@cs.brown.edu

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

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

ABSTRACT

Feature-oriented software designs capture many interesting notions of cross-cutting, and offer a powerful method for building product-line architectures. Each cross-cutting feature is an independent module that fundamentally yields an *open* system from a verification perspective. We describe desiderata for verifying such modules through model checking and find that existing work on the verification of open systems fails to address most of the concerns that arise from feature-oriented systems. We therefore provide a new methodology for verifying such systems. To validate this new methodology, we have implemented it and applied it to a suite of modules that exhibit feature interaction problems. Our model checker was able to automatically locate ten problems previously found through a laborious simulation-based effort.

Categories and Subject Descriptors

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

General Terms

Design,Verification

Keywords

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

*Research partially supported by NSF grants ESI-0010064, ITR-0218973, and CCR-0132659 and by the Brown University UTRA program.

†Current affiliation: The University of Texas at Austin.

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

1. INTRODUCTION

Aspect-oriented programming promises to cleanly capture cross-cutting concerns [30]. Designs based on cross-cutting concerns offer many software engineering benefits, such as separation of concerns, simplified design evolution, and ease of maintenance [9, 10, 22, 37, 38]. Such approaches will not yield a comprehensive development methodology, however, without adequate support for verification at all levels.

A verification framework for cross-cutting designs should support two activities: it must support first proving properties of individual features, and then that composition with other features does not violate these properties. The latter check, however, should be done *compositionally*, i.e., without re-verifying them on the composed system. This problem is challenging because cross-cutting fundamentally generates *open systems*: ones whose components interact with variables defined in other components. Unfortunately, characteristics of cross-cutting software designs render the existing techniques for verifying open systems inadequate.

This paper presents a compositional methodology for verifying open systems arising from cross-cutting concerns. We use the more structured form of cross-cutting proposed in work by Batory [7] and Ossher and Tarr [37], amongst others, and henceforth call these *features*. In particular, we present a significant enhancement of our prior work on verifying feature-oriented designs [23] that handles the numerous verification obstacles arising from open systems.

To motivate and validate our work, we present it in the context of a specific problem: the verification of an email product line application, with an emphasis on detecting *feature interaction* problems. Robert Hall of AT&T Labs designed and analyzed this suite [24]. His analysis was simulation-based, and required substantial human intervention. We attempted to reproduce ten feature interaction errors from his study; our tool successfully detected all ten. Of these, we located only three using the methods described in our prior work. This paper focuses on the techniques that enabled us to locate the other seven.

Section 2 outlines the email suite. Section 3 motivates why feature-oriented designs yield open systems and explains why existing open systems approaches are inadequate. Section 4 describes our enriched model. Section 6 uses the new methodology to compositionally detect feature interactions in the email suite. Section 7 summarizes our perspective and concerns about model checking as a viable tool for compositional reasoning about features. Section 8 discusses prior and related work, while Section 9 presents concluding remarks and future directions.

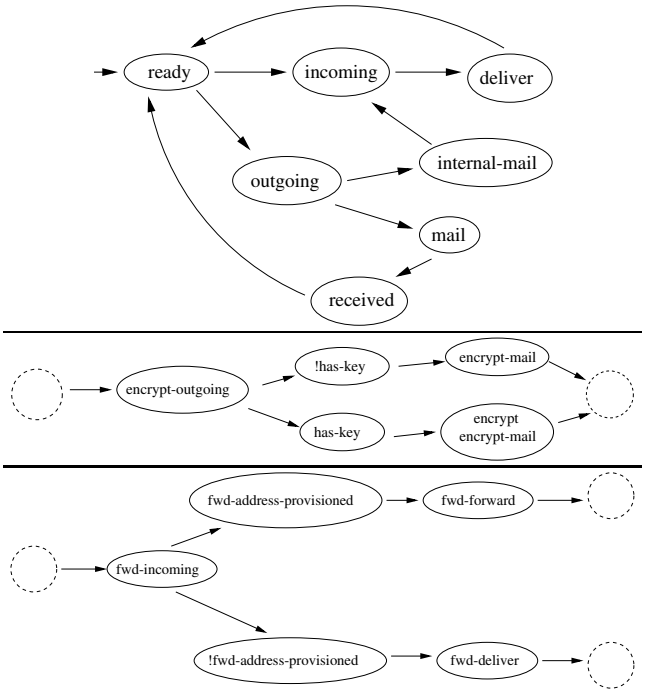

Figure 1: Three features: the base feature, encryption, and forwarding. Dashed states resolve with concrete states during feature composition.

2. A MOTIVATING SCENARIO

We motivate features, their interactions, and why they lead to open systems using an email application as a case study. The example we present is originally due to Robert Hall of AT&T Labs [24]. The application supports eight features (Figure 1): message signature, forwarding, anonymous remailing, encryption, decryption, signed message verification, auto-response, and message filtering.

The following properties, elicited by Hall, should hold of a system containing these features. The properties are stated both in English and in the temporal logic CTL. CTL formulas describe properties of states of a system. A CTL operator consists of two designators: a path quantifier (A for all paths or E for some path) and a temporal operator (G for at all times, F for at some future time, U for until, and R for release, the dual of until). Rather than reproduce the formal semantics [20], we provide three examples of CTL formulas and their English interpretations.

- AFφ says "on all paths φ is true at some future state".
- AGφ says "on all paths, φ is true in all states (i.e. φ is true in all reachable states).
- E[φ U ψ] says "there exists a path on which φ is true in every state until ψ becomes true (ψ must be true in some state along the path).

The properties refer to propositions deliver and retrieved for sending mail. Deliver indicates a message that reaches the current user, while retrieved indicates a message that was mailed to an external user and reaches the recipient.

1. Once a message is signed, the sender field is not altered until the message is delivered or retrieved: AG (sign-msg → A[sender-unchanged U (deliver ∨ received)])
2. When a message is ready to be remailed, it is never mailed out with the sender's identity exposed: AG (wantsRemail → A[anonymous R ¬mail])
3. If one tries to verify a signature, then the message must be verifiable: AG (try-verify → verifiable)
4. When a message is encrypted, it is never sent in the clear: AG (encrypt → A[(deliver ∨ received) R AG ¬(clear ∧ E[¬encrypted U mail])])
5. If a message is to be remailed, it is formatted correctly for the remailer to process it: AG (toRemailer → in-remailer-format)
6. If an auto-response is generated, the response eventually is delivered or retrieved: AG (auto-response → AF (deliver ∨ received))
7. There is no loop where messages are infinitely mailed back and forth: AG AF ready
8. If a message is forwarded, it is eventually delivered or retrieved: AG (forward → AF(deliver ∨ received))

The remainder of the paper refers to these features and properties to illustrate our work.

3. OPEN SYSTEMS AND PRIOR WORK

Consider property 4 of the email application, which states that once a message is encrypted, it is never sent out on the network in the clear. This property holds of the encryption feature. If we compose the encryption feature and the forwarding feature, we will need to check that the forwarding feature preserves this property. The standard CTL model checking algorithm [19] is potentially unsound in this case, however, because the forwarding feature's state machine does not contain the proposition encrypted. *This is not a design error.* Encryption is not part of forwarding, so the forwarding feature should not contain references to the message attributes associated with encryption. This separation of concerns, which underlies feature-oriented design, inherently yields verification tasks involving unknown propositions; unknown propositions lead to open systems.

The existing work in open systems addresses two forms of openness: uncertainty in transitions and ignorance of propositions. Kupferman, Vardi, and Wolper address the former [31]. Their work considers cases in which properties fail due to the values generated by an environment model; their methodology reports a property true of a system only if that property holds regardless of the environment. The work in modal transition systems, similarly, deals with uncertainty of transitions [25]. In contrast, we are concerned with property preservation under specific compositions; most cases of feature interaction arise in contexts where some compositions violate properties and others do not. The Kupferman *et al.* approach is therefore too restrictive for our work.

Bruns and Godefroid consider propositions whose value is unknown; these propositions arise from partial Kripke structures [14]. They employ a 3-valued logic to preserve properties of the partial system in the complete structure. Our work differs in the source of the unknown propositions.

In their work, the unknown propositions arise from considering only a portion of a full state space. In ours, the unknown propositions arise from the *properties* that we wish to verify; the features themselves are closed (by construction) with respect to their propositions. Furthermore, their work does not address a compositional methodology or other open system concerns (such as refinement of propositions and distinctions between control and data propositions) that we motivate in this paper. Our methodology does exploit their algorithm for implementing a 3-valued CTL model checker from an existing 2-valued one [15]. Chechik, Easterbrook, and Devereaux's multi-valued model checker [18] shares the shortcomings of Bruns and Godefroid's work in our context.

The differences between our view of open systems and those in these previous works arise from the models of composition that each work employs. Features encapsulate related portions of a system and compose in a quasi-sequential manner. Open systems in which unknown values arise in the models (rather than from the properties) require another module (the environment) running in parallel to supply the unknown values; Kupferman *et al.*'s work operates in this context. Bruns and Godefroid's work also appears to assume this because their unknown propositions may change value anywhere within a state space (suggesting that the decision of how and when values change is under the control of an external, simultaneously executing entity). In our work, the unknown propositions arise either from data attributes controlled by other features, or from control variables that are local to other features. These differences force us to develop a new methodology for open system verification.

Many researchers have acknowledged the difficulty in detecting feature interactions in the presence of unknown information, and have related this to the frame problem from artificial intelligence [5, 6, 11, 12]. Jackson relates the frame problem to views, which are similar in spirit to cross-cuts [26]. Like Bruns and Godefroid, these techniques all assume a global view of the system, in which all propositions are known in advance. Furthermore, none of their approaches are compositional. Our approach supports the addition of previously unidentified propositions (a higher-level notion of openness) and compositional reasoning.

4. MODELING AND VERIFYING FEATURES AS CLOSED SYSTEMS

Our goal is to develop a compositional methodology for verifying features as open systems. One especially beneficial outcome of such a methodology would be the detection of undesirable feature interactions. As an example, anonymous remailing does not mask a sender's identity if the sender keysigned the message. Other interactions arise from the order in which an application executes features. Although forwarding does not inherently affect encryption, if a message is decrypted prior to forwarding, then a message that had been encrypted goes out on the network in the clear. Such feature interactions are a widespread problem in telecommunications and many other applications, even giving rise to a workshop series. In this paper, we view a feature interaction as undesirable if it violates a formal requirement of either an individual feature or the entire system. We do not discuss the problem of extracting these properties from the requirements.

The main challenges in developing such a methodology are determining what information needs to be included in a feature's interface to support compositional reasoning, and devising techniques to perform these checks. In previous work, we proposed a compositional verification methodology for features that interacted only through sequential transfer of control. The email application involves richer interactions. This section describes our previous model and methodology (for features as closed systems). Section 5 motivates and describes our enriched model and methodology through the email application. A companion paper [34] focuses on the interfaces that support the enriched methodology.

We view a feature as a state machine and feature composition as connecting state machines via transitions specified through interfaces. A state machine model provides a simple and clean abstraction from which to explore verification questions. Each feature (or composition thereof) specifies interfaces (states) where additional features can attach. The following formal definitions from our earlier paper [23] make our model of feature-oriented designs precise. The definitions match the intuition in the figures, so a casual reader may wish to skip the formal definition.

Definition 1. A *state machine* is a tuple $\langle S, \Sigma, \Delta, s_0, R, L\rangle$, where S is a set of states, 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 $\langle\langle exit_1, reentry_1\rangle, \ldots, \langle exit_k, reentry_k\rangle\rangle$ contains a sequence of pairs of states, where each $exit_i$ and $reentry_i$ is a state in machine M_i. State $exit_i$ is a state from which control can enter another feature, 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 features during verification (as motivated throughout the paper).

Definition 3. A *feature* 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 some state with out-degree zero. For each E_i, we call the initial state in_i and the states with out-degree zero *outstates*. These states serve as placeholders for the states to which the feature will connect at composition time. None of these states is in the domain of the labeling function L_i.

Given a base system B, one of its interfaces I, and a feature E, we compose them into a new system by connecting the machines in E to those in B through the states in I, as shown in Figure 2. Definition 4 formalizes our notion of composition; composed designs can 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 [39].

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

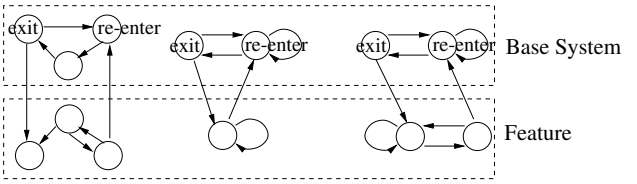

Figure 2: Features, interfaces, and composition

$E_i = \langle S_{Ei}, \Sigma_{Ei}, \Delta_{Ei}, s0_{Ei}, R_{Ei}, L_{Ei} \rangle$ as follows: $S_{Ci} = S_{Mi} \cup S_{Ei} - \{in_i, out_i\}$; $s0_{Ci} = s0_{Mi}$; 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.

Our methodology for verifying properties against individual features can be summarized as follows (full details appear in our prior paper [23]). The methodology currently supports:

1. Proving a CTL property of atomic or composite features (the *verification step*).
2. Deriving *preservation constraints* on the interface states of a feature that are sufficient to preserve each property after composition.
3. Proving that a feature satisfies the preservation constraints of another feature (the *preservation step*). We establish preservation by analyzing only the new feature, not the composition of the two features.

The first activity is challenging when features cross-cut multiple actors, as is standard in practice; the challenge lies in constructing a single state machine corresponding to the feature. 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.

Having constructed a single state machine for a feature, we use the standard CTL model checking algorithm [19] to verify properties of single features. Proving that composition preserves properties is the next challenge. This is where feature-oriented 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 features adds transitions, and thus behaviors, to states in a given module. These extensions are a natural and important part of feature-oriented designs. This characteristic, however, inhibits the use of modular verification techniques based on parallel composition.

Fortunately, modular feature verification reduces to a form of sequential verification. We presented our algorithm in detail in earlier work [23]; Laster and Grumberg proposed a similar algorithm [32]. We summarize the algorithm here in terms of the verification and preservation steps. When

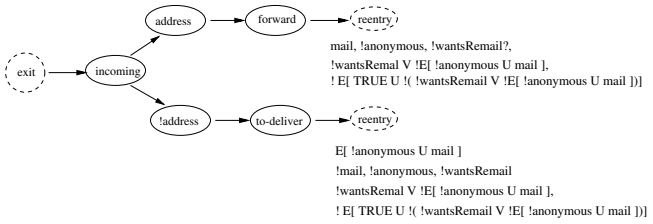

Figure 3: An example of the methodology. The depicted state machine fragment is the filter feature. The dashed states are placeholders for the interface states to which the feature attaches in a larger system. The formulas next to the "re-enter" state are the seeded labels; these labels were copied from a feature with which we composed the filter feature. CTL model checking determines labels on the "exit" state based on the seeded labels.

model checking a property against a feature, we record the labels that the CTL model checking algorithm assigns to the interface states (this is version 1 of the verification step). For the preservation step, when we attach a new feature to those states, we check that the new feature will not invalidate any of those labels. We perform this step by attaching two dummy states to the new feature (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 3). If the derived labels are consistent with the recorded labels, the composition preserves the property of the original feature.

5. MODELING AND VERIFYING FEATURES AS OPEN SYSTEMS

We have used our previous methodology to verify [33] a feature-oriented application called FSATS [8]. The email application that we study in this paper is different from FSATS in several ways:

1. We did not need to model much data in FSATS because the properties concerned control decisions. In contrast, properties in the email example refer to, and crucially depend on, data attributes of the messages.
2. FSATS features are independent, in that they interact only at state transitions, not through shared data.
3. New features in FSATS always attached to the same states in the base system, and never connected to one another. The email application has more of a pipe-and-filter architecture in which features may connect to one another in various orders.

Point 3 doesn't require changes to our prior methodology. Points 1 and 2, however, require enhancements to the methodology, and ultimately our model, for compositional feature verification. Fundamentally, the addition of data attributes leads to interpreting features as open systems. The next two sections explain how this revised interpretation of features affects their verification.

5.1 Unknown Propositions

Using the preservation check on property 4 in the forwarding feature as an example, Section 3 motivated the need to treat features as open systems: to perform this check, we must add the encrypted proposition to the forwarding feature. This proposition captures a data attribute of a mail message that forwarding preserves as it processes the message. Our algorithm cannot assume a concrete truth value for this proposition and remain sound; instead, we must treat this proposition as having unknown value during the check. As 2-valued model checkers treat values as explicitly true or false, we instead use Bruns and Godefroid's 3-valued model checking algorithm [15] for this task.

In 3-valued model checking, propositions can have values {true, false, unknown}. Supporting this requires changes to both the models and the model checker. In the model checker, interpretations of the logical operators change to handle unknown values in a straightforward manner. In the models, the labeling function changes from mapping propositions to {true, false} in each state to mapping propositions to {true, false, unknown} in each state. Accordingly, we augment our definition of a state machine (Defn 1) to contain two labeling functions: one for true propositions and one for false propositions. Propositions not labeled with either true or false in a state are interpreted as unknown.

Definition 5. A state machine $M = (S, \Sigma, \Delta, s_0, R, T, F)$ is a tuple where S is a set of states, Σ and Δ are sets of input and output atomic propositions, $s_0 \in S$ is the initial state, $R \subseteq S \times \mathrm{PL}(\Sigma) \times S$ is the transition relation, $T : S \rightarrow 2^{\Delta}$ indicates which propositions are true in each state, and $F : S \rightarrow 2^{\Delta}$ indicates which propositions are false in each state ($\forall s \in S, T(s) \cap F(s) = \emptyset$).

A 3-valued model checker can return true, false, or unknown as the value of a property in a structure. From a verification perspective, the unknown result is less useful than a true or false result. Techniques for determining concrete truth values in the presence of unknowns are therefore extremely useful. When no proposition maps to unknown in any state, 3-valued model checking reduces to 2-valued model checking and returns either true or false; models with no unknowns are called *complete*. Bruns and Godefroid's algorithm checks each property in two complete models: one in which all unknowns are replaced with true (the *optimistic* model) and one in which all unknowns are replaced with false (the *pessimistic* model). A property is guaranteed to be false if it evaluates to false in the optimistic model, and guaranteed to be true if it evaluates to true in the pessimistic model. If neither of these guarantees hold, their algorithm reports the property as having unknown value.

Our methodology could treat all propositions that arise in the property but are not in the model as unknown during preservation checks, but that is too conservative. Consider property 2, which refers to proposition wantsRemail. This proposition does not capture a data attribute of a message. Rather, it is a *control proposition*: it determines control-flow within a feature. Control propositions of one feature are never true in another feature because features do not execute simultaneously. This lets us set the control propositions from other features to false during model checking. Thus, when the designer can partition the propositions into control and data subsets, our technique can exploit this design information.

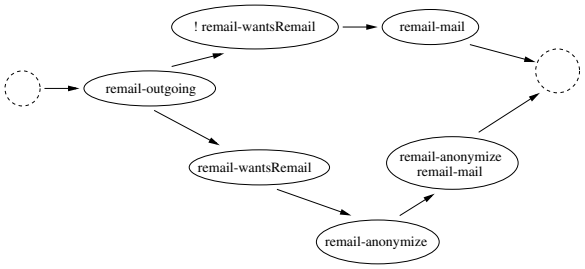

Figure 4: The remail feature.

Given this distinction, we reduce feature-oriented verification to 3-valued model checking as follows:

Verification step, version 2: Given a property and a feature (state machine), consider all propositions that are in the property but not in the feature. For each of these propositions, if it is a control proposition, set it to false in all states of the feature; otherwise, set it to unknown in all states. Use 3-valued model checking to verify the property against the augmented state machine.

We also enhance the preservation step to recognize our use of 3-valued model checking in the verification step. Preservation checks confirm that true properties remain true upon composition. The 3-valued model checking framework reports a property as true only if it evaluated to true in the pessimistic model. Our methodology therefore needs to confirm that the labels generated during the pessimistic check are preserved; the labels generated during the optimistic check are irrelevant. During the verification step, we therefore store the state labelings on the interface states from the pessimistic check in the feature's interface.

Preservation step, version 2: Verifying that a feature F_2 preserves a property φ of a feature F_1 upon composition of F_1 and F_2 entails the following steps:

- As in the original preservation step, extend F_2 with dummy states representing the interface states of F_1.
- Copy the pessimistic labels from the in-states of F_1 to the dummy ending states of F_2 and copy the pessimistic atomic propositions from the out-states of F_1 to the dummy starting states of F_2 (as in Figure 3).
- Set all of the control propositions of F_1 to false (rather than unknown) in every state of F_2; set data proposition of F_1 to unknown in every state of F_2.
- Use 2-valued model checking to verify that all pessimistic CTL labels on the out state of F_1 hold in the dummy starting state of F_2.

5.2 Evolving Propositions

Propositions with unknown values enable the preservation checks required in feature-oriented verification, but are insufficient to enable compositional feature verification. Compositional verification requires that once we verify a property of a feature, we should not need to traverse that feature again during the preservation checks for that property. Feature-oriented systems sometimes require the interpretation of propositions to evolve upon composition; this in turn complicates compositional reasoning. The key point of this section is that open systems arise not only from abstraction

and decomposition (the conventional contexts for open system verification research), but also from system evolution.

Consider property 2, which says that messages passing through the anonymizing remailer cannot reveal information that identifies the sender. How is anonymous defined in this property? From the perspective of the remailer feature alone (Figure 4), anonymous is the same as the proposition remail-anonymize from the remailer. Once we add the signing feature, however, a message also needs to be unsigned in order to be considered anonymous. In other words, adding the signing feature changes the property statement from

AG (wantsRemail → A[remail-anonymize R ¬mail]) to
AG (wantsRemail → A[(remail-anonymize ∧¬signed) R ¬mail]).

How can we verify this property against the remail feature compositionally, when the property might change in unexpected ways upon composition?

We present the approach intuitively before providing the formal details. In our concrete example, evolution logically strengthened anonymous: we replaced remail-anonymize with remail-anonymize ∧¬signed. We can reasonably expect the evolution of propositions to logically strengthen or weaken their previous interpretations (otherwise, one feature would completely override another, which lies outside the scope of our current model). Strengthening and weakening are defined as follows: if $expr$ and $expr'$ refer to the original and evolved interpretations of a proposition, $expr'$ strengthens $expr$ if $expr' \equiv expr \wedge augment$, and $expr'$ weakens $expr$ if $expr' \equiv expr \vee augment$, for some expression $augment$.

Suppose we had verified the original property in the remail feature, then needed a preservation check for this property in the signing feature. What labels would we copy from the remail feature to the dummy states of the signing feature? Since the sign feature changes the property, we cannot assume that the labels from the original verification remain valid. When propositions evolve, therefore, our technique from the previous section is not applicable.

Assume for the moment that we had anticipated that a future feature might place additional restrictions on (*i.e.* strengthen) anonymity. We could have verified the property AG (wantsRemail → A[(remail-anonymize ∧ augment) R ¬mail]) against the remail feature. The labels stored in remail for a preservation check would therefore be valid for any extension that strengthened the definition of anonymity. To verify the formula containing augment against the remail feature, however, would require 3-valued model checking since the interpretation of augment is unknown inside the remailer (by construction). This example outlines our proposed methodology for handling evolving propositions. We will verify properties under the assumption that certain propositions may be strengthened or weakened, then use the labels arising from those assumptions to perform preservation checks. While this approach will not let us perform all composition checks compositionally, it should let us perform many checks in that manner.

This proposal raises several concerns. *Does a user need to know all the features and propositions before beginning verification?* No, our technique is designed to support design evolution, including the addition of unexpected features. If an extension re-interprets a proposition that the designer had not expected to evolve, some existing features may need to be re-verified. *Does failure of an augmented property in the verification step yield useful feedback?* Our algorithm actually verifies each property in both its original and augmented forms to help identify the actual conditions under which a property fails. *Wouldn't multiple augmented propositions in one property greatly reduce the likelihood of meaningful verification results?* Yes, but we have not seen that case frequently in practice; furthermore, our approach is analogous to Bruns and Godefroid's optimistic and pessimistic interpretations on this point. In short, we believe the full algorithm, which we now present, adequately addresses these concerns within the limits of software engineering practice.

Both the verification step and the preservation step must change to handle evolving propositions. First, we need to distinguish between propositions whose interpretations may evolve (henceforth called *evolving propositions*) and those whose interpretation will remain fixed. We leave this distinction to the modeler. We extend the model checker with an additional input, a re-mapping function R from evolving propositions to boolean expressions over non-evolving propositions. When the model checker encounters an evolving proposition p, it evaluates $R(p)$; non-evolving propositions are evaluated directly. We now present the revised verification and preservation steps.

Verification step, version 3: Given a property φ to verify of a feature F_1 under a re-mapping R, verify φ under three interpretations, storing the labels arising from each check separately in F_1's interface:

1. The standard 3-valued check of φ using R.
2. A strengthening check in which each evolving proposition p in φ is strengthened to $R(p) \wedge augment$ for some new proposition $augment$.
3. A weakening check in which each evolving proposition p in φ is weakened to $R(p) \vee augment$ for some new proposition $augment$.

Given a feature F_2 that must preserve property φ already proven of feature F_1, we must choose one of these three sets of labels to use in the preservation check. The choice depends on whether the interpretation of evolving propositions in F_2 strengthens or weakens those in F_1, as detailed in the following algorithm.

Preservation step, version 3: Let R_1 be the re-mapping used to verify φ in F_1 and let R_2 be the new re-mapping associated with F_2.

- If $R_2(p)$ strengthens $R_1(p)$ for all evolving propositions p in φ, check whether the pessimistic strengthened case held in F_1. If so, extend F_2 with dummy interface states as described in Section 5.1, but copy/confirm the pessimistic interface labels that arose under the strengthened interpretation. If not, there is no need to proceed with verification of F_2 because F_1 already violates the property.
- If $R_2(p)$ weakens $R_1(p)$ for all evolving propositions p in φ, follow the previous case using pessimistic weakened in place of pessimistic strengthened.
- If $R_2(p)$ is logically equivalent to $R_1(p)$ for all evolving propositions p in φ, extend F_2 with dummy interface states as described in Section 5.1, but copy/confirm the pessimistic interface labels that arose under interpretation R_1.

- In all other cases, re-verify φ against F_1 using R_2, then apply version 2 of the preservation algorithm to check preservation in F_2.

5.3 Soundness

The soundness of this methodology arises from a combination of the soundness of the methodology for verifying features as closed systems, the soundness of Bruns and Godefroid's 3-valued checking with optimistic and pessimistic interpretations, and the logic of strengthening and weakening. Due to lack of space, we defer the formal soundness proof to a full version of the paper.

6. EMAIL CLIENT CASE STUDY

To evaluate the effectiveness of our interfaces and methodology for compositionally verifying feature-oriented systems, we searched for feature interactions in the email application described in Section 2 (which Hall's work showed to be rich in feature interactions [24]). We wanted to determine

- whether we could detect feature interactions compositionally given our interfaces and methodology,
- the extent to which each enrichment to our methodology contributed to detecting actual interactions, and
- whether interactions can be detected through combining small numbers of features.

The last point is important because the number of features involved in an interaction points to the overhead that our methodology would incur from the compositional preservation tests. We have not proposed techniques for identifying likely interactions; this is an important and difficult, yet orthogonal, problem. In this case study, we are more concerned with the potential overhead of our methodology with respect to the amount of interface information needed to perform the preservation tests compositionally.

The email features as shown in this paper do not appear to cross-cut the system, since we describe each feature through a single state machine (cross-cutting arises in our framework from the separate machines for individual actors in Defn. 2). The full email system does involve multiple actors (the users and the network, for example). We use single actor versions in this case study because the distinction between single and multi-actor features is irrelevant for the study of interfaces that we undertake here.

Our experiments use a model checker that we built specifically for handling our feature-oriented verification methodology. The checker supports the methodologies presented in this paper and in our previous work. We do not present performance figures here because the state machines for these models are too small to generate meaningful performance figures. As the goal of this case study is to test the *utility* of our models and methodology for detecting feature interactions, we view performance as an orthogonal problem for this paper. Future papers will consider larger case studies designed to test our theory for performance.

Each of the properties from Section 2 held when verified against the feature that was mainly responsible for implementing it, but failed upon composition with other features.[1] Furthermore, all evolving propositions were strictly either weakened or strengthened during re-mapping; we never needed to re-verify a property already proven of a feature after re-mapping.

Table 6 summarizes the feature interactions that we detected using our modeling and verification methodology. The table summarizes the property whose violation led to the undesired interaction, the (ordered) composition of features with which we detected the interaction, a description of the undesirable interaction, and an indication of what part of our verification methodology detected the interaction. The table shows some interesting results: first, our original (Section 4) methodology (with no re-mapping or 3-valued model checking) detected only three of the interactions. This suggests that while the overhead and additional verification expense of our richer model is not always needed for detecting feature interactions, the richer methodology is crucial for many interactions.

The feature interactions arising from property 4 are unique because they required a minimum of three interacting features composed in a certain order. Fewer features yielded no interaction: the property holds of the encryption feature alone, the property holds when composing decryption with encryption because the decryption feature doesn't mail messages, and the property holds of both encryption-autorespond and encryption-forward because the message stays encrypted. Different compositions of all three features, however, expose the undesirable interaction arising from this property:

- The property doesn't hold when autorespond or forward is composed onto the encryption-decryption compound feature because this composition introduces a path from a state where the message is clear (and stays clear) to mail. A 3-valued check exposes this.
- The property doesn't hold when decryption follows encrypt-autorespond or encrypt-forward. The proposition clear is weakened from false to false $\vee$ decrypt-successful. A pessimistic weakened check on encrypt-autorespond or encrypt-forward exposes this.

Checking property 4 requires 3-value checks because encrypted and decrypt-successful are data propositions. After composition with forwarding, for example, the property still optimistically holds (because it assumes a message is encrypted before it gets forwarded). Under a pessimistic check, however, encrypted is false throughout the forwarding feature, which violates the property.

7. PERSPECTIVE ON VERIFICATION

Identifying verification techniques that provide good support for feature-oriented verification is an interesting and important open problem. Both our previous work and the work reported here use model checking as the underlying verification technology. Model checking is a reasonable first choice: its automated nature allows us to prototype methodologies quickly and easily, and its low-level nature has forced us to identify fine-grained details about the feature interfaces needed to support compositional verification. Although model checking is not necessarily a natural choice for software verification, many research efforts are now exploring how well it applies to this domain.

Our choice of model checking has clearly affected our models of features and their interfaces: in particular, interfaces would likely not associate labels with states were we

[1] For the rest of this section, we will implicitly assume that features are composed with the email base (Figure 1) prior to verification, which defines the propositions mail and deliver.

Property	Features Composed (in execution order)	Yields Problem	Under Re-Mappings	Verification Techniques
1	sign, forward	Sender field changes if message is signed and then forwarded	re-map sender-unchanged from true to ¬forward	Evolving Propositions
1	sign, remail	the remailer changes the sender field	re-map sender-unchanged from true to ¬anonymize	Evolving Propositions
2	sign, remail	signing a message gives away the identity even if the sender is changed	re-map anonymous from anonymize to anonymize ∧ ¬signed	Pessimistic Strengthened
3	encrypt, verify	encrypting a signed message hinders sign-verification. Attempts to decrypt after sign-verification may also fail.	re-map verifiable from true to true∧¬encrypted	Pessimistic Strengthened
4	encrypt, decrypt, forward	a message can be encrypted, mailed, decrypted, and forwarded in the clear	re-map clear from false to ¬decrypt-successful	3-valued check
4	encrypt, decrypt, autorespond	a message can be decrypted and autoresponded where the response contains the body of the original message.	re-map clear from false to ¬decrypt-successful	3-valued check
5	encrypt, remail	the remailer cannot decipher encrypted messages to determine the recipients.	strengthen in-remailer-format from true to true ∧ ¬encrypt	Pessimistic Strengthened
6	autorepond, filter	the filter feature can potentially discard messages generated by the autoresponse feature		Previous
7	forward, remail	forwarding from a user to his pseudonym creates an infinite cycle of mailings.		Previous
8	forward, filter	filter can discard forwarded messages		Previous

Table 1: Summary of undesirable feature interactions and the modeling and verification techniques needed to detect them. Numbers in the property column refer to the numbered descriptions of the properties from Section 2.

not using state machine models and CTL model checking. Nonetheless, our experiences using model checking in the context encourage us to reflect on how viable model checking will be as a foundation for feature-oriented verification.

First, the amount of interface information that compositional model checking of features seems to require is an immediate concern. We currently store labels on several interface states for checks under both strengthening and weakening of evolving propositions. This information becomes less useful as the number of evolving propositions in a property increases. We also store partitions into control and data variables. Multi-actor features require even more interface information [23]. Although the interface information has not proven excessive in this study, it could become so in a larger application that contains hundreds of features spanning multiple actors. Additional case studies are required to determine when the overhead of our interfaces outweighs the benefits of compositional feature verification.

Next, features interact implicitly through data. A viable model of feature interaction therefore must support modeling and reasoning about data. Model checkers' limitations in reasoning about data are well known: the main problem is the combinatorial explosion in propositions needed to encode data values as booleans. Many model checking efforts handle this problem through a combination of abstraction and cone-of-influence reduction. Given the deep co-mingling of control and data in both the models and properties of some feature-oriented systems, we are unsure whether these approaches will be useful in this context. In many cases, the design methodology inherently performs a partial abstraction because a feature only contains the propositions that are relevant to it.

We believe that the real problem lies in the need to arguably overspecify data in most state-based specifications. For data-intensive domains such as this one, declarative specifications (as employed by Alloy [27]) are likely more viable in the long term. Effective integration of declarative specifications into model checking or other feature-oriented verification techniques remains an open problem.

Finally, our work on feature-based verification suggests that CTL is a more viable logic than LTL for compositional reasoning about features. This contradicts the conventional wisdom which cites LTL as better suited to compositional reasoning [41]. This departure reflects the difference in composition semantics between our work, which supports a form of sequential composition, and most compositional model checking, which supports parallel composition.

8. RELATED WORK

Numerous pipe-and-filter style models of feature composition have been proposed, with Zave and Jackson's Distributed Feature Composition [28] being one of the most notable. Our models of features and their interactions have arisen from our goal to support compositional verification of feature-oriented designs. To the best of our knowledge, other models of feature composition have not been designed with associated verification methodologies. Furthermore, our full methodology supports features over multiple actors, which distinguishes our work from other pipe-and-filter style approaches, including Zave and Jackson's.

Other verification researchers have discussed methodologies for reasoning under sequential composition [1, 2, 21, 32]. These efforts differ from ours in many ways: none handle open systems, none were created towards supporting cross-cutting design methodologies, and none support coordination of multiple actors.

This paper has presented a form of detection for feature interaction problems, which have received substantial attention in the software engineering literature [29, 42]. Our detection of feature interactions through compositional reasoning and our handling of multiple actors distinguish our work from other approaches to modeling or detecting feature interaction through model checking [3, 4, 13, 28, 29]; Section 3 discussed other work on features as open systems.

Several recent efforts have applied model checking to reasoning about aspect-like constructs. Chechik and Easterbrook reason about compositions of concerns using multi-valued model checking [17]. Their framework helps identify which concern (feature) is responsible for property violations when checking composed systems; it does not address how to prove properties through compositional reasoning on individual concerns. Ubayashi and Tamai propose a method of applying model checking to programs written using AspectJ [40]. Their verification methodology, however, is simply to weave the program together and verify the result. They leave any notion of compositional verification to future work. Nelson, Cowan and Alencar [36] perform a case study on reasoning about cross-cutting concerns. They employ two tools, Alloy and the LTS checker, to reason about both component and emergent properties. They too, however, compose the concerns into a single global specification in lieu of defining a compositional verification model. Lin and Lin also present a temporal logic-based approach to reasoning about feature interactions, but their approach is not compositional [35].

9. CONCLUSIONS AND FUTURE WORK

Verification methodologies for cross-cutting constructs are an important open problem in software engineering. Indeed, by increasing the expressive power of programming notations, these constructs generate an even greater need for verification, and as such cannot be considered mature development idioms in the absence of verification methods. We present a methodology that supports compositional reasoning about features and their interactions. Our features span multiple actors, support shared data across features, and provide a realistic preliminary model for verifying systems of cross-cutting concerns.

Compositional feature verification inherently requires reasoning about open systems. We show that existing approaches to open systems in verification are based on assumptions that do not apply in the context of feature-oriented design. We then build a verification methodology for features as open systems using three-valued model checking as an underlying verification technology. In order to be both compositional and useful for detecting feature interactions in practice, our methodology requires separation of propositions into control and data variables and a notion of redefining propositions within properties as new features are added to a system. Three-valued model checking is instrumental in supporting both of these requirements.

Using our methodology, we have analyzed an email application that is rich in feature interaction problems. We attempted to uncover ten known feature interactions using our methodology; we detected them all reasoning compositionally (rather than reasoning about the entire composed system which contained the undesirable interactions), using all of the modeling and verification techniques that comprise our approach. This validates our derived feature interfaces as useful for supporting compositional feature interaction detection.

In future work, we plan to look into verification methodologies other than model checking that might offer better support for reasoning about unknown data values; we are specifically interested in exploring declarative specifications in this context. We also plan to conduct further case studies on different designs known to display feature interactions, such as the Bellcore Benchmark [16].

Acknowledgments. We thank Robert Hall for generously providing detailed help with his email suite and the anonymous reviewers for their extensive comments.

10. 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] C. Areces, W. Bouma, and M. de Rijke. Description logics and feature interaction. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics*, pages 28–32, 1999.

[4] C. Areces, W. Bouma, and M. de Rijke. Feature interaction as a satisfiability problem. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications Systems*. IOS Press, 2000.

[5] E. Astesiano and G. Reggio. A discipline for handling feature interaction. In *Requirements Targeting Software and Systems Engineering*, number 1526 in Lecture Notes in Computer Science, pages 95–119. Springer-Verlag, 1998.

[6] P. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 153–167. IOS Press, 1997.

[7] D. Batory. Product-line architectures. In *Smalltalk and Java Conference*, Oct. 1998.

[8] 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.

[9] 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.

[10] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, Oct. 2001.

[11] J. Blom, R. Bol, and L. Kempe. Automatic detection of feature interactions in temporal logic. Technical Report DoCS 95/61, Department of Computer

Systems, Uppsala University, 1995.

[12] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.

[13] K. Braithwaite and J. Atlee. Towards automated detection of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 36–59. IOS Press, 1994.

[14] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *International Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 274–287. Springer-Verlag, 1999.

[15] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *International Conference on Concurrency Theory*, number 877 in Lecture Notes in Computer Science, pages 168–182. Springer-Verlag, 2000.

[16] E. Cameron, N. Griffeth, Y. Lin, M. Nilson, W. Schnure, and H. Velthuijsen. A feature interaction benchmark for IN and beyond. In *Feature Interactions in Telecommunications Systems*, pages 1–23. IOS Press, 1994.

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

[18] M. Chechik, S. M. Easterbrook, and B. Devereux. Model checking with multivalued temporal logics. In *Proceedings of the International Symposium on Multiple Valued Logics*, 2001.

[19] 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.

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

[21] 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.

[22] 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.

[23] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Symposium on the Foundations of Software Engineering*, Sept. 2001.

[24] R. J. Hall. Feature interactions in electronic mail. In *Feature Interactions in Telecommunications Systems*. IOS Press, 2000.

[25] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *European Symposium on Programming*, 2001.

[26] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4), Oct. 1995.

[27] D. Jackson. Alloy: a lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, Feb. 2000.

[28] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, Oct. 1998.

[29] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, Oct. 1998.

[30] 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.

[31] O. Kupferman, M. Vardi, and P. Wolper. Module checking. In *International Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 75–86. Springer-Verlag, 1998.

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

[33] H. Li, K. Fisler, and S. Krishnamurthi. The influence of software module systems on modular verification. In *9th International SPIN Workshop on Model Checking of Software*, number 2318 in Lecture Notes in Computer Science, pages 60–78. Springer-Verlag, 2002.

[34] H. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *IEEE International Symposium on Automated Software Engineering*, 2002.

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

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

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

[38] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[39] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, 1998.

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

[41] M. Y. Vardi. Branching vs. linear time: Final showdown. Invited talk, European Symposium on the Theory and Practice of Software (ETAPS). Available at `http://www.cs.rice.edu/~vardi/papers/index.html`, 2001.

[42] P. Zave. Calls considered harmful and other observations: A tutorial on telephony. In T. Margaria, editor, *Second International Workshop on Advanced Intelligent Networks*, 1997.