Modular Verification of Open Features Through Three-Valued Model Checking*

Harry C. Li†
Brown University, USA

Shriram Krishnamurthi
Brown University, USA

Kathi Fisler
WPI, USA

February 14, 2003

Abstract

Feature-oriented programming organizes programs around features rather than objects, thus better supporting extensible, product-line architectures. Programming languages increasingly support this style of programming, but programmers get little support from verification tools. Ideally, programmers should be able to verify features independently of each other and use automated compositional reasoning techniques to infer properties of a system from properties of its features. Achieving this requires carefully designed interfaces: they must hold sufficient information to enable compositional verification, yet tools should be able to generate this information automatically because experience indicates programmers cannot or will not provide it manually. We present a model of interfaces that supports automated, compositional, feature-oriented model checking. To demonstrate their utility, we automatically detect the feature-interaction problems originally found manually by Robert Hall in an email suite case study.

1 Introduction

Modules are crucial to large-scale software construction [Par72]. Modules divide a system into coherent collections of data structures and functionality that programmers can assemble into a suite of services. The benefits that modules bestow, such as independent development and code reuse, have ensured the widespread adoption of modules in software development.

Having different developers write the modules in a system increases the likelihood of incompatibility between modules. Programmers therefore need some level of composition verification to protect against latent errors that are not detected until late into development or even deployment. Type checking at module boundaries is perhaps the most basic and widespread form of verification. Each module's interface specifies its services as a series of function or method names and the type signatures on their inputs and outputs; the module also specifies the interfaces it expects of the modules with which it will eventually compose. Type checkers confirm that an individual module satisfies its own interface and that it uses services from other modules type-correctly. Modern languages such as ML [MTH90] and Java [GJS96] support this basic notion of modular verification, and it is so useful and convenient that programmers use it daily without complaint.

While type-based modular verification is a handy first line of defense, it proves only a very simple theorem (typically, that well-typed programs will not go "wrong" [Mil78]); furthermore, this theorem is fixed and built into the type system. Developers often need to prove richer theorems about a system's behavior. Behavioral verification can uncover subtle errors such as concurrency violations, race conditions, deadlock, and progress failures. As programs grow more complex, and increasingly use communication and concurrency, behavioral verification grows more critical.

The feasibility of modular behavioral verification is unfortunately diminished by a simple but critical practical concern: the need for specifications. While programmers voluntarily write types, decades of experience have shown that programmers are highly unlikely to write more complex specifications of a module's behavior. This problem persists even when these specifications are fed to tools that can provide concrete feedback [FL01]. Worse, programmers often

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

†Current affiliation: University of Texas at Austin.

simply lack sufficient understanding of the program's behavior and may not have the training necessary to correctly use the specification logics. Without specifications, however, the modular verification tools cannot function, leaving the programmers who most need verification unable to exploit it.

One tempting proposition is to compose a complete program out of the modules, then verify the program as a whole. Verifying the entire program, however, has several shortcomings. First, not all modules are available at the same place, because they are written by independent authors and assembled (in a componential fashion [Szy98]) by a client. Second, even when the modules are available (say at the client), the total number of system configurations can be too numerous: for instance, in a product line construction [CN02], the total number of combinations of product line features can exhibit combinatorial explosion. Finally, even a single one of those configurations may be too large to verify en masse due to the well-known problem of state explosion [CGP00].

For behavioral verification to be useful and tractable in practice, it must therefore apply to modules, rather than only to whole programs. Ideally, a *modular* verification methodology should support proving properties about individual modules and inferring properties of composed systems from properties of the individual modules; furthermore, these methods should retain the automation of type checking. Most importantly, given a behavioral property expected of a whole system, the technique must automatically generate the module specifications because programmers often will not, and sometimes may not be able to, supply them. This is the essence of automated software engineering: to automatically handle tasks that programmers cannot, or will not, perform manually.

The verification technique that this paper defines specifically addresses feature-oriented modules. These modules encapsulate individual program features that cross-cut systems [KLM+97] and contain state-machine representations of code fragments that implement a feature's functionality for each actor in the overall system. In recent years, researchers from a variety of applications areas have noted that programming with cross-cutting concerns can simplify a variety of software engineering problems such as maintenance, evolution, and product-line development [BO92, CN02].

This paper focuses on the interfaces that feature-oriented modules need in order to support modular model checking of behavioral properties. Interfaces must contain sufficient information for tools to prove whether composition would violate the properties proven of an individual module. This requires interfaces to contain constraints, similar to verification conditions, that other modules must satisfy at composition time. Our methodology derives these conditions *automatically* during feature verification. Thus, for feature-oriented modules we are able to lift the benefits of automated modular verification to the level of behavioral properties.

This paper also demonstrates the utility of our interfaces through a case study, which we use as a running example throughout the paper. The case study is based on an analysis of an email system originally conducted by Robert Hall [Hal00]. This example is interesting because it contains a substantial number of feature interactions; in our methodology, these manifest as properties that hold of individual features, yet fail after composition. Hall originally identified these interactions manually. Using our methodology, we can detect these interactions automatically and *compositionally* given desired properties of the individual features.

Section 2 provides an overview of the case study used in this paper. Section 3 presents an overview of and prior work on open systems. Section 4 describes our core approach to feature-oriented verification in the context of closed systems. Section 5 extends our approach to open systems. Section 6 presents the results of our case study. Section 8 reviews related work. Section 9 offers concluding remarks.

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 [Hal00]. The application offers several features, a characteristic of product line systems; these features can, however, adversely interact with one another in many ways. The application provides a database which stores information pertinent to individual users, such as their encryption keys, mail aliases, and forwarding addresses (if any). The application contains the following features (Figure 1 shows some of their state machines): basic mail delivery, digital signatures, forwarding, anonymous remailing, encryption, decryption, signature verification, auto-reply, filtering (based on sender's hostname), and mail hosting. The features connect through a pipe-and-filter style architecture [SG96].

The following properties, elicited by Hall, should hold of a system containing these features. 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.

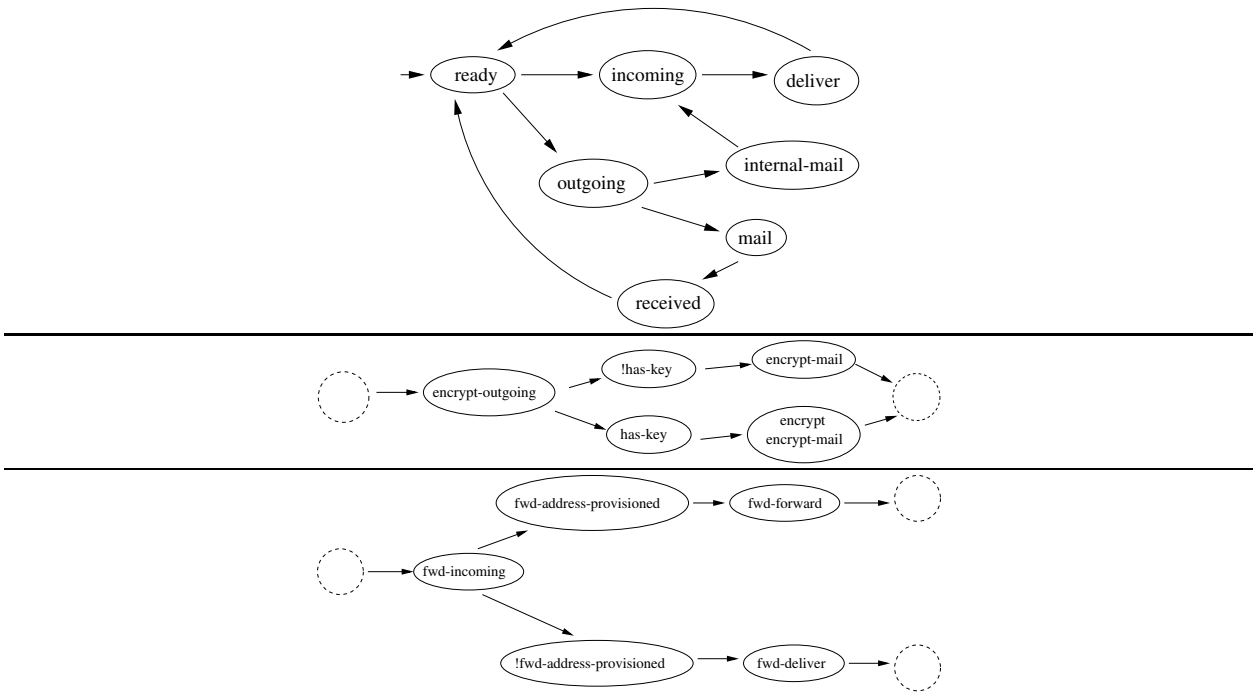


Figure 1: Three features: the base feature, encryption, and forwarding. Dashed states unify with concrete states during feature composition.

The properties are stated both in English and in the temporal logic CTL [CES86, CGP00]. 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 all times, F for some future time, U for until, and R for release, the dual of until). Rather than reproduce the formal semantics, 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).

Hall found a variety of interactions by manually inspecting numerous configurations of these features. Many of these interactions violate straightforward requirements on the individual features; this paper studies ten of these requirements. We state the requirements both informally and in CTL.

1. Once a message is signed, the sender field is not altered until the message is delivered or received.
   Formula: AG[sign-msg $\rightarrow$ A[sender-unchanged U (deliver $\vee$ received)]]
2. When a message is ready to be remailed, it is never mailed out with the sender's identity exposed.
   Formula: AG[wantsRemail $\rightarrow$ A[anonymous R $\neg$mail]]
3. If a receiver tries to verify a signature, then the message must be verifiable.
   Formula: AG[try-verify $\rightarrow$ verifiable]
4. When a message is encrypted, it is never decrypted and then sent in the clear.
   Formula: AG[encrypt $\rightarrow$ A[(deliver $\vee$ received) R AG $\neg$(decrypted $\wedge$ E[$\neg$encrypted U mail])]]
5. If a message is to be remailed, it is formatted correctly for the remailer to process it.
   Formula: AG[toRemailer $\rightarrow$ in-remailer-format]

6. If an auto-response is generated, the response eventually is delivered or received.
   Formula: AG[auto-response → AF (deliver ∨ received)]

7. There is no loop where messages are infinitely mailed back and forth.
   Formula: AG AF ready

8. If a message is forwarded, it is eventually delivered or received.
   Formula: AG[forward → AF (deliver ∨ received)]

9. If the auto-responder replies to a message, then that message's subject line must be in the clear.
   Formula: AG[auto-response-incoming → clear]

10. If an outgoing message is signed, then its body is never changed unless is it delivered or retrieved.
    Formula: AG[sign-mail ∧ signed → A[delivered ∨ retrieved R body-unchanged]]

11. If a mailhost generates an error message, then that message is eventually retrieved or delivered.
    Formula: AG[mailhost-errorMail → AF (deliver ∨ received)]

Each of these properties holds in the feature that implements it. Each property also fails when the feature that implements it is composed with another (specific) feature. Section 6 describes these interactions and the specific aspects of our methodology that detect the failures.

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 [CES86] 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 [KVW98]. 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 [HJS01]. 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 [BG99]. 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 [BG00]. Chechik, Easterbrook, and Devereaux's multi-valued model checker [CED01] shares the shortcomings of Bruns and Godefroid's work from the perspective of this work.

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. Hall classifies several types of interactions; ours fall into his "Type II" classification. Some researchers have related feature interaction detection to the frame problem from artificial intelligence [AR98, AA97, BBK95, BMR95]. Jackson relates the frame problem to views, which are similar in spirit to cross-cuts [Jac95]. 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 key-signed 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.

4.1 Modeling Features and Their Compositions

Our formal model of feature-oriented systems views each feature as a single state machine. Our previous work [FK01] shows how to reduce models where each feature has multiple state machines to the single-machine model. Hence we adopt the single-machine model here for simplicity.

Definition 1 Let ϕ be a set of atomic propositions. $PL(\phi)$ denotes the set of propositional logic expressions over the set of variables in ϕ.

Definition 2 A state machine is a tuple $M = (S, \Sigma, \Delta, s_0, R, Tr, Fa)$ 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,
- $R \subseteq S \times PL(\Sigma) \times S$ is the transition relation,
- $Tr : S \rightarrow 2^{\Delta}$ indicates which propositions are true in each state, and $Fa : S \rightarrow 2^{\Delta}$ indicates which propositions are false in each state ($\forall s \in S, Tr(s) \cap Fa(s) = \emptyset$).

This is the standard definition of a state machine, augmented with distinct labeling functions for true and false labels instead of just one labeling function for the true labels. This distinction supports our use of 3-valued model checking.

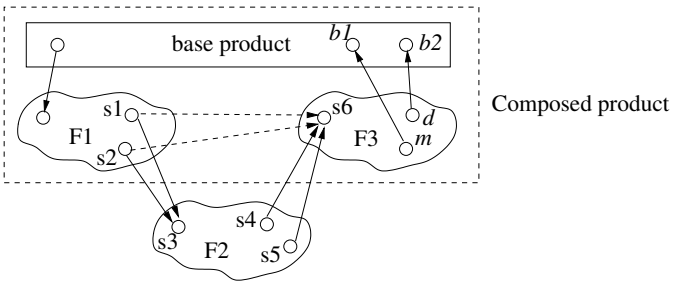


Figure 2: Illustrations of features, products, and their compositions.

We expect features in pipe-and-filter product-line systems to compose in a chain, where the chain begins and ends with some basic infrastructure that is common to all products within the family (such as basic mail delivery, in the email case study). A composition of features and the base infrastructure forms a *product*, where a product consists of both a state machine and a set of interfaces where new features may be inserted into the system. We capture the common infrastructure in a *base product*, which is like a core feature. When we verify individual features, we must do so within the context of the base product so that we can establish properties of the feature relative to the common infrastructure.

Definition 3 A *base product* is a state machine $(S, \Sigma, \Delta, s_0, R, Tr, Fa)$ with an interface $\langle\{s_{\text{outgoing}}\}, S_{\text{connect}}\rangle$ where

- $s_{\text{outgoing}} \in S$,
- $S_{\text{connect}} \subset S$,
- R contains an edge from s_{outgoing} to each state in S_{connect} (this represents the system with no features).

Definition 4 A feature is a state machine $M = (S, \Sigma, \Delta, s_0, R, Tr, Fa)$ with an interface $\langle\{s_{\text{incoming}}\}, S_{\text{exit}}, R_{\text{exit}}\rangle$ where

- $s_{\text{incoming}} \in S$ is the initial state for the feature,
- $S_{\text{exit}} \subseteq S$ is the set of exit states; these states must have out-degree 0.
- $R_{\text{exit}} \subseteq S_{\text{exit}} \times PL(\Sigma) \times 2^{\Delta} \times 2^{\Delta}$ specifies constraints on edges from exit states to states in the eventual composed system. The last two arguments in each tuple specify true and false sets of propositions on the states to which these edges are expected to connect when integrating the feature into a product.

Our model builds up products by iteratively composing new features into products. In this framework, adding a feature to a product yields another product. We could define compositions of features into new features, but do not present those details here to simplify the presentation. The extension to feature compositions outside the context of products is, however, straightforward.

In order to connect a feature to a product, we need to match up the states in S_{connect} of the base system with the specifications in the partial transitions from the states in S_{exit} in the feature; if a state has no labels, we view it as satisfying all specifications. Intuitively, we will add an edge from a state in S_{exit} to each state in S_{connect} that satisfies the specification on the true and false propositions given in R_{exit} from the exit state. The following two definitions make this precise.

Definition 5 Let e_1 and e_2 be states in (possibly different) features. e_1 and e_2 *unify* if $Tr(e_1) = Tr(e_2)$ and $Fa(e_1) = Fa(e_2)$ (i.e., if both states agree on their true and false labels), or if one of the states has no labels in Tr or Fa.

Definition 6 Let P be a product with state machine $(S_P, \Sigma_P, \Delta_P, s_0, R_P, Tr_P, Fa_P)$ and interfaces

$$\{\langle S_{\text{outgoing}_1}, S_{\text{connect}_1}\rangle, \ldots, \langle S_{\text{outgoing}_k}, S_{\text{connect}_k}\rangle\}.$$

Let F be a feature with state machine $(S_F, \Sigma_F, \Delta_F, s_{0_F}, R_F, Tr_F, Fa_F)$ and interface $\langle S_{\text{incoming}}, S_{\text{exit}}, R_{\text{exit}}\rangle$. The composition of P and F via interface $\langle S_{\text{outgoing}_i}, S_{\text{connect}_i}\rangle$ is a product P_C. The state machine component of P_C is $(S_C, \Sigma_C, \Delta_C, s_0, R_C, Tr_C, Fa_C)$ where

- $S_C = S_P \cup S_F$,
- $\Sigma_C = \Sigma_P \cup \Sigma_F$,
- $\Delta_C = \Delta_P \cup \Delta_F$,
- $R_C = R_P \cup R_F \cup R_{\text{new}}$ *except* all edges between the interface states S_{outgoing_i} and S_{connect_i} from R_P. R_{new} contains the following edges:
  - All (s, pl, s'') such that there exists $(s, pl, s') \in R_P$ such that $s \in S_{\text{outgoing}}$, $s' \in S_{\text{connect}}$, $s'' \in S_{\text{incoming}}$ and s' and s'' unify.
  - All (s, pl, s') such that $s \in S_{\text{exit}}$, $(s, pl, t, f) \in R_{\text{exit}}$, $s' \in S_{\text{connect}}$, $t \subseteq Tr_P(s')$ and $f \subseteq Fa_P(s')$.
- $Tr_C = Tr_P \cup Tr_F$, and
- $Fa_C = Fa_P \cup Fa_F$.

The interfaces of P_C is the set of interfaces from P except $\langle S_{\text{outgoing}_i}, S_{\text{connect}_i}\rangle$ and augmented with two new interfaces:

- $\langle S_{\text{outgoing}_i}, S_{\text{incoming}}\rangle$
- $\langle S_{\text{exit}}, S_{\text{connect}}\rangle$

Figure 2 illustrates these definitions more intuitively. The figure shows a product consisting of a base product and two features F_1 and F_3, and the composition of feature F_2 onto this product (after F_1). The composition is performed via an interface $\langle\{s_1, s_2\}, \{s_6\}\rangle$. The interface on F_2 is $\langle\{s_3\}, \{s_4, s_5\}, \emptyset\rangle$. Composition removes the dashed edges and adds the four edges that connect F_2 to the product. The italic labels b_1 and b_2 on the states of F_3 and the base product capture the idea behind states unifying on composition. When F_3 was composed with the base product, edges connected the states from F_3 to the base based on matching up the labels.

4.2 The Core Verification Methodology

Our verification methodology entails three tasks:

1. Proving a CTL property of an individual feature through model checking (the *verification step*).
2. Automatically deriving *preservation constraints* on the interface states of features and products that help detect feature interactions compositionally.
3. Checking whether a feature F and a product P satisfy one another's preservation constraints at composition time (the *preservation step*). We establish preservation by analyzing at most F or P individually, not the composition of F and P.

We derive the preservation constraints during the verification step using (a variant of) CTL model checking. The standard CTL algorithm works by labeling all states with subformulas of the property to be verified. When we verify a property against a feature (in the verification step), the interface states are labeled with some of these subformulas. During the preservation step, we must check whether the overall product violates any of these labels. We therefore store these labels in the interface and check whether they still hold during preservation checks. More formally, the verification step operates as follows:

Verification step, version 1: Let F be a feature with interface I and B be the base product for the product-line that can contain F. Let φ be a CTL property to prove against F. Compose F and B via I into an extended feature F'. Use CTL model checking to verify φ in the initial (incoming) state from F'.

The formal model of features and their compositions from Section 4.1 is *constructive*, in that it captures the details needed to define and compose products and features. The verification step suggests that interfaces also have an *analytic* component, where we store data required for compositional verification. A refined notion of interfaces therefore accompanies each version of the verification step.

Definition 7 (*Interfaces, version 1*) Each interface I of a feature or a product contains a mapping from states in the interface to a set of CTL properties (the labels placed on those states during the verification step).

We give an intuitive description of the preservation step by referring to Figure 2. When we compose F_2 into the rest of the product (containing the base, F_1, and F_3), we will need to perform two sets of checks: first, that the labels on s_4 and s_5 of F_2 hold once those states transition to F_3; second, that the labels on s_1 and s_2 hold once those states transition to s_3. We perform each set of checks by augmenting each of the feature and the product with dummy interface states from the other, assuming certain labels on the dummy states, and verifying the remaining labels on the dummy states through model checking. The following description formalizes this intuition.

Preservation step, version 1: Let F be a feature with interface I_F and P be a product. Let I_P be the interface of P through which we intend to compose F and P. The algorithm checks for two possible sources of interactions:

1. (Prove that P doesn't interfere with properties of F) We must check whether every label stored on a state in S_{exit} of I_F still holds once F is connected to P. Add a dummy state s_d to P with an edge from s_d to each state in S_{connect}. For each state s in S_{exit} and each label φ on s in I_F, copy the propositional labels from s to s_d, then model check φ at s_d. Report an error iff one of these verification fails.

2. (Prove that F doesn't interfere with properties within P) We must check whether the labels stored on states in S_{outgoing} of I_P still holds once F is connected to P. Add one dummy state to F for each state in the interface I_P and insert edges between F and the dummy states that match those added when composing F with P. Copy the labels from S_{connect} to their respective dummy states (which are reachable from F) and copy the propositional labels from states in S_{outgoing} to their respective dummy states (which reach F). For each formula φ that labels a state s in S_{outgoing}, model check the formula in the dummy state corresponding to s. Report an error iff one of these verifications fails.

Although the preservation step may appear expensive, in practice we rarely need model checking to confirm that P preserves the properties of F. Most labels on the exit states of F are simple labels such as "the mail state is reachable", i.e., they refer explicitly to reachability conditions on the states in the base product. In this case, we can simply check these reachability constraints once when composing a feature into the system, thereby amortizing these checks across multiple product compositions.

The correctness of this approach to preservation follows from our earlier work [FK01]. The rest of this paper will revise these core algorithms to handle issues that arise in the context of open systems.

5 Modeling and Verifying Features as Open Systems

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 need a 3-valued model checker. We used Bruns and Godefroid's 3-valued model checking algorithm [BG00] in this paper; Chechik, Easterbrook and Devereaux's multi-valued model checker would also apply [CED01].

In 3-valued model checking, propositions can have values {true, false, unknown}; this explains our use of separate true and false labeling functions in Definition 2. Propositions not labeled with either true or false in a state are

interpreted as unknown. We use the symbol $\leq$ to denote an ordering on the precision of values in 3-valued logic: unknown $\leq$ true and unknown $\leq$ false, while leq does not relate true and false.

In a 3-valued model checker, interpretations of the logical operators extend to unknown values in a straightforward manner. 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 [BG00]. If neither of these guarantees hold, their algorithm reports the property as having unknown value. Figures 3 through 5 illustrate verification of the forwarding feature under each of the regular, optimistic, and pessimistic interpretations (the "normalized form" in the figures rewrite all formulas without universal temporal operators).

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, which increases the likelihood of a concrete result from the model checker. Thus, when the designer can partition the propositions into control and data subsets, our technique can exploit this design information.

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

Verification step, version 2: Let F be a feature and φ a property to prove of F. For each proposition p that is in φ but not in the labeling functions of F, set p to false in all states of F if p is a control proposition; otherwise set p to unknown in all states. Use 3-valued model checking to verify the property against the augmented state machine. Store the labels arising from the pessimistic and optimistic checks separately in F's interface.

Definition 8 (*Interfaces, version 2*) Each interface I of a feature or a product contains:

- A set of propositions from the product or feature which correspond to control propositions.[1]
- A mapping from states in the interface to a set of CTL properties true during the optimistic check.
- A mapping from states in the interface to a set of CTL properties true during the pessimistic check.

We must enhance the preservation step in accordance with our use of 3-valued model checking in the verification step. When using 2-valued model checking, preservation checks confirm that true properties remain true upon composition. In a 3-valued setting, the preservation step uses two sets of checks: one with the optimistic labels (which will detect property violations) and one with the pessimistic labels (which could confirm property preservation). If neither the optimistic nor pessimistic checks result in concrete answers, we must analyze the property against the composed system to determine its status in the composed system. Within the limits of our case study, however, the optimistic and pessimistic checks were always sufficient.

Preservation step, version 2: The preservation step proceeds as in version 1, with two modifications. For each property label φ to be checked:

1. For each proposition p in φ but not in the model being checked, set p to false (if a control proposition) or unknown (if a data proposition) in all states of the model.
2. Run the preservation algorithm twice: once copying the pessimistic labels, and once copying the optimistic labels. If the property fails under the optimistic labels, report an error. If the property holds under the pessimistic labels, report the property as preserved.

[1] This set can be an underapproximation without sacrificing soundness.

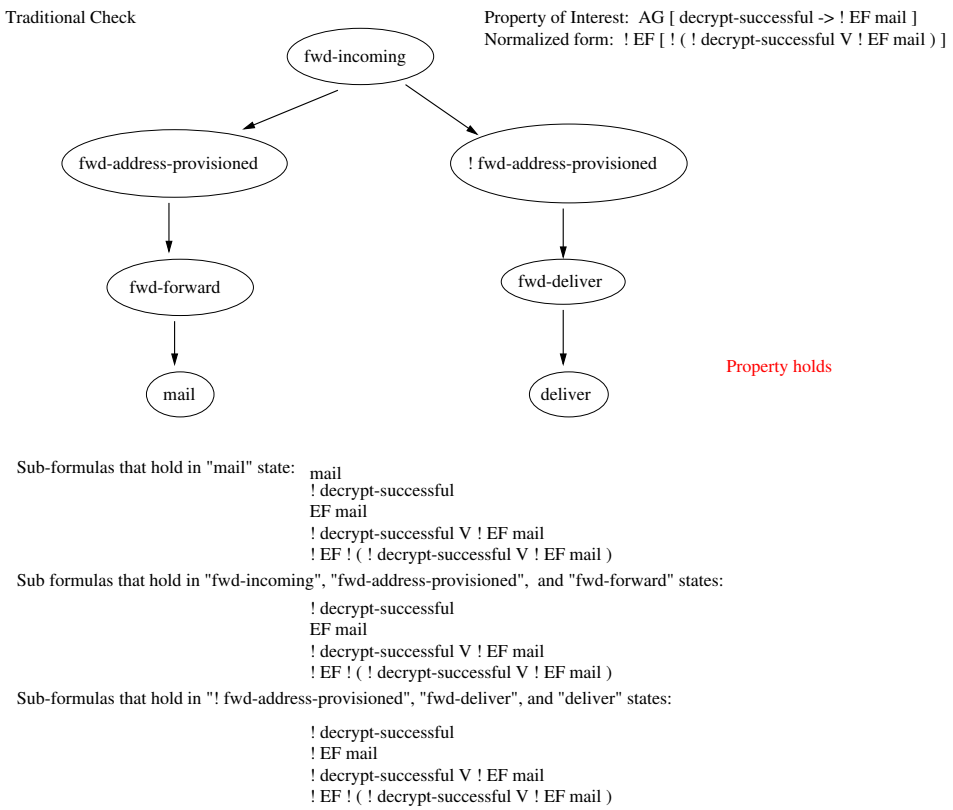
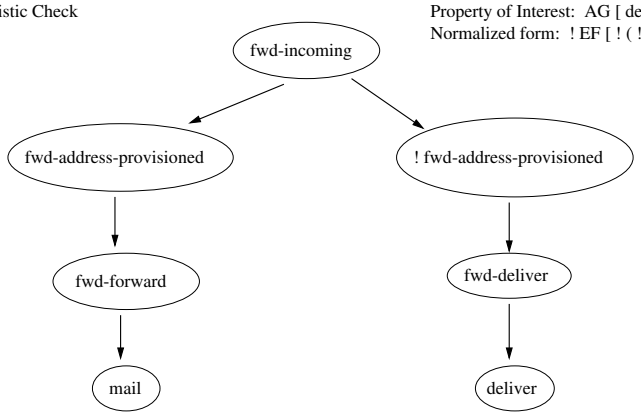


Figure 3: Verifying the forwarding feature: the plain verification results.

Optimistic Check

Property of Interest: AG [decrypt-successful -> ! EF mail]
Normalized form: ! EF [! (! decrypt-successful V ! EF mail)]



Property holds

Same as traditional check because the optimistic check assumes decrypt-successful is false everywhere

Sub-formulas that hold in "mail" state:

mail
! decrypt-successful
EF mail
! decrypt-successful V ! EF mail
! EF ! (! decrypt-successful V ! EF mail)

Sub formulas that hold in "fwd-incoming", "fwd-address-provisioned", and "fwd-forward" states:

! decrypt-successful
EF mail
! decrypt-successful V ! EF mail
! EF ! (! decrypt-successful V ! EF mail)

Sub-formulas that hold in "! fwd-address-provisioned", "fwd-deliver", and "deliver" states:

! decrypt-successful
! EF mail
! decrypt-successful V ! EF mail
! EF ! (! decrypt-successful V ! EF mail)

Figure 4: Verifying the forwarding feature: the optimistic verification results.

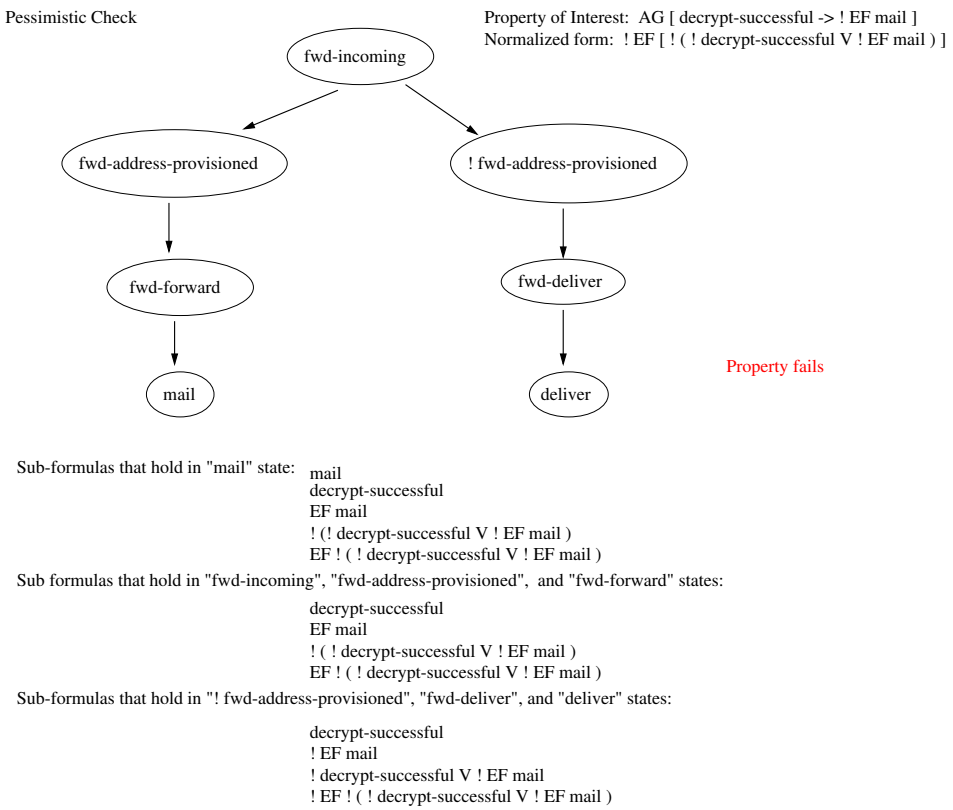


Figure 5: Verifying the forwarding feature: the pessimistic verification results.

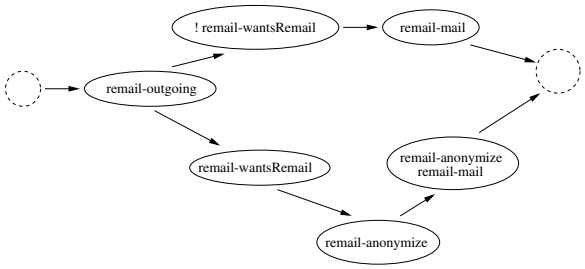


Figure 6: The remail feature.

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 6), 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

$$\mathsf{AG}\ (\mathsf{wantsRemail} \rightarrow \mathsf{A}[\mathsf{remail\text{-}anonymize}\ \mathsf{R}\ \neg\mathsf{mail}]) \text{ to}$$
$$\mathsf{AG}\ (\mathsf{wantsRemail} \rightarrow \mathsf{A}[(\mathsf{remail\text{-}anonymize} \wedge \neg\mathsf{signed})\ \mathsf{R}\ \neg\mathsf{mail}]).$$

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 $\wedge\neg$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:

Definition 9 Let $expr$ and $expr'$ be boolean expressions. $expr'$ 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 $\rightarrow$ A[(remail-anonymize $\wedge$ augment) R $\neg$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; the method remains sound as long as the set of evolving propositions is over-approximated.

We extend the model checker with an additional input, an *interpretation function* R from evolving propositions to boolean expressions over non-evolving propositions: we use notations $M, s, R \models \varphi$ and $M, s, R \not\models \varphi$ to denote properties being true and false (respectively) in this extended model checker (we do not use a particular notation for a model check returning unknown). When the model checker encounters an evolving proposition p, it evaluates $R(p)$; non-evolving propositions are evaluated directly. We lift the definition of strengthening and weakening to interpretations, then present revised verification and preservation steps.

Definition 10 An interpretation R_2 *strengthens* interpretation R_1 iff for each proposition p in the domain of R_1, either $R_1(p) = R_2(p)$ or $R_2(p)$ strengthens $R_1(p)$. If R_2 strengthens R_1 and $R_2(p) \neq R_1(p)$ for any proposition p, then we say that R_2 *strictly strengthens* R_1. We define *weakens* and *strictly weakens* at the level of interpretations analogously.

Verification step, version 3: Given a property φ to verify of a feature F under a interpretation R, perform version 2 of the verification step three times, each under a different interpretation:

1. A 3-valued using R. If this check fails, the property fails to hold and the algorithm stops.
2. A strengthening check (denoted $\models_{\mathrm{ST}}$) in which each evolving proposition p in φ is strengthened to $R(p) \wedge augment_p$ for some new proposition $augment_p$. If a check fails, record false as the result of verification in the label sets for that check.
3. A weakening check (denoted $\models_{\mathrm{WK}}$) in which each evolving proposition p in φ is weakened to $R(p) \vee augment_p$ for some new proposition $augment_p$. If a check fails, record false as the result of verification in the label sets for that check.

Each of the different interpretations gives rise to different sets of labels during the verification step. The interfaces must expand to store all of these labels accordingly, as well as the core interpretation that was in effect when the properties were verified.

Definition 11 (*Interfaces, version 3*) Each interface I of a feature or a product contains:

- The control propositions of the product or feature.[2]
- A mapping from states in the interface to a set of CTL properties true during the optimistic check.
- A mapping from states in the interface to a set of CTL properties true during the pessimistic check.
- A mapping from states in the interface to a set of CTL properties true during the optimistic strengthened check.
- A mapping from states in the interface to a set of CTL properties true during the pessimistic strengthened check.
- A mapping from states in the interface to a set of CTL properties true during the optimistic weakened check.
- A mapping from states in the interface to a set of CTL properties true during the pessimistic weakened check.

[2] This set can be an underapproximation without sacrificing soundness.

Although these interfaces appear to be getting rather complex, it is important to remember that a designer needs only supply the constructive interface and the partition into control and data propositions. All of the labels are generated and stored automatically. In addition, the sets of labels and the labels themselves will tend to be small, so the space overhead is not as severe as Definition 11 might suggest.[3]

In order to complete our formal model, each product and feature must contain an interpretation of its evolving propositions. In the case of a product, this interpretation evolves as additional features are added to the product. To formalize this notion, we must define the composition of interpretations. When composing interpretations, the values from features override those from products, regardless of whether the feature strengthens or weakens the interpretation in the product. This may seem counterintuitive, with a better strategy to keep the stronger interpretation in the composition. Such an interpretation, however, would not allow adding a feature to weaken an interpretation within a product. This situation arises in the case of anonymous being weakened to anonymous $\vee$ signed. If would make no sense to lose this more general interpretation in the overall product, especially since the new interpretation arises from a composition of features. Having the feature's interpretation override conflicts in the product captures the idea that products evolve in accordance with their feature sets.

Definition 12 Let F and P be a feature and a product to compose. Let R_F be the interpretation associated with F and R_P be the interpretation associated with P. For all propositions p in the domain of both R_P and R_F, assume $R_F(p)$ either strengthens or weakens $R_P(p)$. We define the composition R_C of R_F and R_P to be an interpretation where

- For all propositions in the domain of R_P but not in the domain of R_F, $R_C(p) = R_P(p)$,
- For all other propositions, $R_C(p) = R_F(p)$.

R_C is undefined if there exists a proposition p such that $R_F(p)$ neither strengthens nor weakens $R_P(p)$ (this definition allows $R_F(p) = R_P(p)$ since strengthening and weakening can be vacuously satisfied with $augment$ values of true and false, respectively).

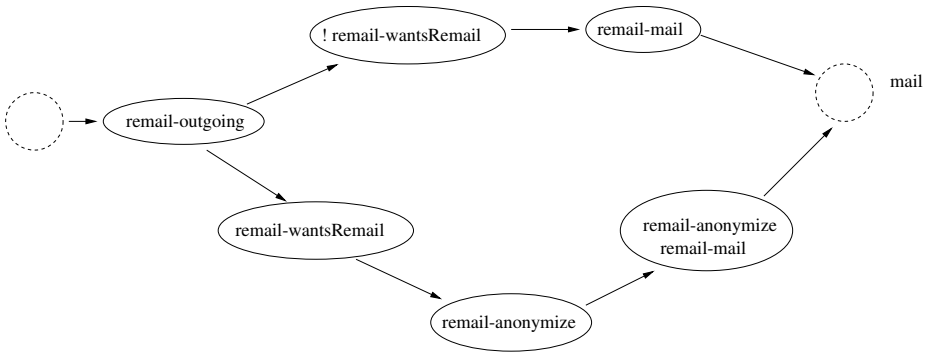
Naturally, the preservation step must also account for strengthening and weakening. Version 2 of the preservation step performs checks in two directions: one analyzing labels of the product against the feature and one analyzing labels of the feature against the product. We parameterize the following definition of the preservation step over the one being analyzed and the one whose labels are being confirmed, with their respective interpretations. the interpretation R_A of the one being analyzed and the interpretation R_C of the one whose labels are being confirmed.

Preservation step, version 3: For each check in version 2 of the preservation step, let φ be the label being confirmed, A be the feature/product being analyzed and let C be the feature/product whose labels are being confirmed. Let R_A and R_C be the interpretations of A and C, respectively. Choose the labels to copy to the interface states according to the following algorithm:

- If $R_A(p)$ strengthens $R_C(p)$ for all evolving propositions p in φ,
  - If the optimistic strengthened labels map to false (indicating that the optimistic strengthened check failed), report an error.[4]
  - Otherwise, perform version 2 of the preservation step using the strengthed versions of the (optimistic and pessimistic) labels. If version 2 reports a concrete answer, return it.
- If $R_A(p)$ weakens $R_C(p)$ for all evolving propositions p in φ,
  - If the optimistic weakened labels map to false (indicating that the optimistic weakened check failed), report an error.
  - Otherwise, perform version 2 of the preservation step using the weakened versions of the (optimistic and pessimistic) labels. If version 2 reports a concrete answer, return it.

[3] Note that each set is at most linear in the size of each property, which itself tends to be quite small.

[4] Counterexample generation is one of the benefits of model checking. If we want to retain compositional counter-example generation, we can store the counterexample traces in the interface whenever a strengthened or weakened check produces false; this stored information would be sufficient to reconstruct a counterexample in the composed model without verifying in the composed model.



Property of Interest: AG [remail-wantsRemail -> A (anonymous R ! mail)]
Normalized Form: ! EF ! [! remail-wantsRemail V ! E (! anonymous U mail)]
Sub-formulas: remail-wantsRemail, anonymous, mail
! remail-wantsRemail, ! anonymous
E (! anonymous U mail)
! E (! anonymous U mail)
! remail-wantsRemail V ! E (! anonymous U mail)
! (! remail-wantsRemail V ! E (! anonymous U mail))
EF ! [! remail-wantsRemail V ! E (! anonymous U mail)]
! EF ! [! remail-wantsRemail V ! E (! anonymous U mail)]

Strengthened results: anonymous = remail-anonymize AND unknown
property fails
trace: remail-outgoing -> remail-wantsRemail -> remail-anonymize ->
remail-anonymize, remail-mail -> mail

Weakened results: anonymous = remail-anonymize OR unknown
property holds
labels on remail-mail: ! remail-wantsRemail, ! anonymous, E (! anonymous U mail),
! remail-wantsRemail V ! E (! anonymous U mail), ! EF ! [! remail-wantsRemail V ! E (! anonymous U mail)]
labels on remail-anonymize, remail-mail: anonymous, ! remail-wantsRemail, ! E (! anonymous U mail),
! remail-wantsRemail V ! E (! anonymous U mail), ! EF ! [! remail-wantsRemail V ! E (! anonymous U mail)]

Figure 7: Verifying the remailing feature with strengthening and weakening.

- If $R_A(p)$ is logically equivalent to $R_C(p)$ for all evolving propositions p in φ, follow version 2 of the preservation step with the regular (non-strengthened or weakened) labels. (Note: if the two interpretations are logically equivalent, this algorithm would first attempt the strengthened and weakened tests, performing this check only if neither of those produced a concrete answer. For efficiency, we could modify the conditions for strengthened and weakened tests to require at least one proposition to strictly strengthen or weaken; this would not affect our soundness theorems.)
- In all other cases, or if none of the previous cases yields a concrete answer, re-verify φ against C using R_A, then apply version 2 of the preservation algorithm (with the new labels) to check preservation in A.

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. Our methodology is not complete due to a combination of our use of 3-valued logic and strengthening and weakening interpretations.

Intuitively, our soundness theorems state that if the compositional methodology reports a particular property label as being true or false at a state, then model checking the same property on the corresponding state of the composed system would yield the same result. Our soundness results do not make claims about cases where the compositional methodology yields unknown as the result of a verification. We present the argument as two separate theorems, one concerning labels on states from the feature, and the other concerning labels on states from the product.

Our soundness proofs rely on an argument about the soundness of preservation checks. We perform preservation checks by attaching dummy states to a state machine; these dummy states represent the interface states to which the state machine will be connected during composition. We copy property labels from the interface to these dummy states, and verify properties in this augment state machine. Intuitively, we claim that any property that labels a state in the augmented machine also labels the corresponding state in the composed machine.

We claim that the labels on the states of the fragment and the dummy initial state are identical to those that would appear had we verified the property against a composed system. This claim is valid because state labels are determined by the labels of their successors in CTL model checking, and composition does not add paths from the end of a fragment back to its initial state. The following lemma formalizes this argument

Lemma 1 *(The preservation lemma) Let M be a state machine that will be composed with state machine S via interface I. Let M' be M augmented with a dummy state for each state in I, with edges between states of M and the new states in M' determined by the definition of composition (Definition 6). For all dummy interface states that serve as sinks of M', copy all labels from the corresponding states in I to the states of M'; for dummy interface states that serve as sources of M', copy all propositional labels from the corresponding states in I to the states of M'. Let s be any state in M' and let φ be a CTL property. Model checking φ at state s in M' returns the same value as model checking φ at the state corresponding to s in the composition of M and S.*

Proof: This lemma follows from the definition of CTL model checking. CTL model checking determines the labels on a state from the labels of its successor states. Thus, the lemma holds as long as composing M and S cannot affect the labels on the sink interface states. The labels on the sink interface states can only change if composition changes the set of states reachable from the interface states. Since our composition model prohibits edges that create new cycles at composition time, the labels on the sink interface states must be preserved upon composition. The lemma therefore holds.

□

Our results also depend on Bruns and Godefroid's theorems that any formula that is true under the pessimistic interpretation is true in the full model and any formula that is false under the optimistic interpretation is false in a full model [BG00]. We do not duplicate their theorem statements in this paper.

Lemma 2 *Let S be a state machine, φ be a CTL formula, s be a state in S, and R_1 and R_2 be interpretations of evolving propositions in φ. Assume R_2 strengthens R_1. Then $S, s, R_1 \models_{\mathrm{ST}} \varphi \leq S, s, R_2 \models \varphi$ (i.e., the result of model checking φ under R_2 is at least as precise as the result of the strengthening model check under R_1). If R_2 strictly strengthens R_1, then $S, s, R_1 \models_{\mathrm{ST}} \varphi = S, s, R_2 \models \varphi$.*

Proof: Interpretations affect model checking only at the level of propositions because interpretations map propositions to boolean expression over other propositions. It is therefore sufficient for us to argue that the theorem holds for all propositional formulas; the inductive definition of CTL model checking naturally lifts this result to properties in all of CTL.

The propositional proof breaks into several cases:

- Assume φ is a proposition p. If p is not an evolving proposition, then the theorem holds because the value of p at s is determined by the state labeling in S and is not affected by R_1 or R_2. If p is an evolving proposition, then its truth value at s is that of the expression that p maps to under the corresponding interpretation. We therefore need to consider the relationship between $R_1(p)$ and $R_2(p)$.
- If $R_1(p) = R_2(p)$, then the strengthening model check will check $R_1(p) \wedge augment_p$ while the regular model check will confirm $R_1(p)$. If $R_1(p)$ is false, then both the regular and strengthening model checks will return false for p at s. If $R_1(p)$ is not false (unknown or true), then since $augment_p$ has value unknown by definition, the strengthening check will return unknown while the regular check will return $R_1(p)$. In both cases, the theorem holds.
- Otherwise, $R_2(p) = R_1(p) \wedge augment_p$. In this case, the strengthening model check used $R_1(p) \wedge augment_p$ in place of p. This is equivalent to $R_2(p)$ up to renaming between $augment$ and $augment_p$. Since these variables are logically equivalent (interpreted as unknown), the result of model checking both expressions on s is equivalent. The theorem therefore holds in this case.

The only case in which the two model checks did not return the same result was when $R_1(p) = R_2(p)$, in which case R_2 does not strictly strengthen R_1. The strictly strengthening clause in the theorem therefore holds.

□

Corollary 1 *If the model checks performed in Lemma 2 are both pessimistic, then $S, s, R_1 \models_{\mathrm{ST}} \varphi = S, s, R_2 \models \varphi$ regardless of whether R_2 strengthens or strictly strengthens R_1.*

Proof: This follows immediately from the argument in the proof of Lemma 2.

□

Lemma 3 *Let S be a state machine, φ be a CTL formula, s be a state in S, and R_1 and R_2 be interpretations of evolving propositions in φ. Assume R_2 weakens R_1. Then $S, s, R_1 \models_{\mathrm{WK}} \varphi \leq S, s, R_2 \models \varphi$ (i.e., the result of model checking φ under R_2 is at least as precise as the result of the weakening model check under R_1). If R_2 strictly weakens R_1, then $S, s, R_1 \models_{\mathrm{WK}} \varphi = S, s, R_2 \models \varphi$.*

Proof: The proof is analogous to that for Lemma 2.

□

Corollary 2 *If the model checks performed in Lemma 3 are both optimistic, then $S, s, R_1 \models_{\mathrm{WK}} \varphi = S, s, R_2 \models \varphi$ whether R_2 weakens or strictly weakens R_1.*

Lemma 4 *Let R_F and R_P be interpretations and let R_C be the composition of R_F and R_P. If R_F strengthens (resp. weakens) R_P for all propositions in the domain of both R_F and R_P, then R_C strengthens (resp. weakens) R_F for all propositions in the domain of R_F.*

Proof: This follows trivially from the definition of R_C, since $R_C(p) = R_F(p)$ for all propositions p in the domain of R_F.

□

Lemma 5 *Let R_F and R_P be interpretations and let R_C be the composition of R_F and R_P. If R_F strengthens (resp. weakens) R_P for all propositions in the domain of both R_F and R_P, then R_C strengthens (resp. weakens) R_P for all propositions in the domain of R_P.*

Proof: By Lemma 4, R_C strengthens (weakens) R_F for all propositions in the domain of R_F. R_C therefore strengthens (weakens) R_P for all propositions in the domain of both R_F and R_P by transitivity. For all propositions p in the domain of R_P and not in the domain of R_F, $R_C(p) = R_P(p)$, so the result holds trivially.

□

We now present the main soundness result as two separate theorems. Version 1 of the preservation step consists of two main subparts: one for determining whether the the product interferes with the properties of the feature, and one for determining whether the feature interferes with the properties of the product. We handle each case in a separate soundness theorem.

Theorem 1 *Let P be a product and F be a feature. Let P_C be the composition of P and F via interface $I = \langle S_{\text{outgoing}}, S_{\text{connect}} \rangle$. Let s_f be a state in S_{incoming} of F and let φ be a CTL formula that labels s_f (in one of the various sets of interface labels). Let R_F be the interpretation in use when φ was verified against F. Let R_P be the interpretation for P and let R_C be the composition of R_F and R_P. If the preservation step reports that φ is preserved when composing P and F via I (i.e. that P does not interfere with properties of F—part 1 of the preservation step, version 1), then $P_C, s_f, R_C \models \varphi$. If the preservation step reports that φ is violated when composing P and F via I, then $P_C, s_f, R_C \not\models \varphi$.*

Proof: By Definition 12, R_P neither strengthens nor weakens R_F. Two cases therefore exist:

- R_P and R_F are logically equivalent on all propositions in φ. According to the preservation step algorithm (version 3), we use the non-strengthened or weakened labels and apply version 2 of the preservation step. Version 2 reports φ as holding (failing) if it holds (fails) in the pessimistic (optimistic) interpretations. Our theorem therefore reduces to the soundness of using pessimistic (optimistic) models to determine truth (falsehood) in a 3-valued model. Bruns and Godefroid's theorem establishes this soundness.

- The previous case did not hold, in which case we re-verify P using R_F and then use version 2 of the preservation step to check φ. Following reverification, this case reduces to the previous one, in which R_P and R_F are logically equivalent on all propositions in φ.

□

Theorem 2 *Let P be a product and F be a feature. Let P_C be the composition of P and F via interface $I = \langle S_{\text{outgoing}}, S_{\text{connect}} \rangle$. Let s_p be a state in S_{outgoing} of P and let φ be a CTL formula that labels s_p (in one of the various sets of interface labels). Let R_P be the interpretation from P, R_F be the interpretation from F, and R_C the composition of R_P and R_F. If the preservation step reports that φ is preserved when composing P and F via I (i.e. that F does not interfere with properties of P—part 2 of the preservation step, version 1), then $P_C, s_p, R_C \models \varphi$. If the preservation step reports that φ is violated when composing P and F via I, then $P_C, s_p, R_C \not\models \varphi$.*

Proof: If state s_p does not reach any state from F in P_C, then the theorem holds trivially because CTL model checking determines property labels from the properties of its successors. If s_p reaches any state in F, then it must reach a state in the interface S_{incoming} of F. By the preservation lemma (Lemma 1), the theorem holds for s_p if every label on every state s_i in S_{incoming} of F during the preservation check is a valid label on s_i in P_C. Another application of the preservation lemma reduces this to proving that all labels on states s_e in S_{exit} of F during the preservation step are still valid on s_e in P_C. We therefore consider the statement only for these states.

Let s_e be a state in S_{exit} and let ψ be a label on s_e in F'. We must prove that ψ labels s_e in P_C. As this theorem concerns the preservation step (version 3), the proof breaks into cases depending upon the relationship between R_F and R_P. In this theorem, F is the system being analyzed (called A in the preservation step) and P is the system being confirmed (called C in the preservation step).

- If R_F strengthens R_P for all evolving propositions in ψ and the optimistic strengthened labels map to false, then the algorithm reports that ψ does not hold in the composed system. The soundness of this step follows from the soundness of false results under optimistic models predicting false results in full 3-valued models. Bruns and Godefroid's theorem completes the proof in this case.

  A similar argument covers the case when the optimistic weakened labels map to false.

- Assume R_F strengthens R_P for all evolving propositions in ψ but the optimistic strengthened labels do not map to false. The algorithm performs version 2 of the preservation step using the strengthened versions of the labels. Assume version 2 reports that $F', s_e, R_F \models \psi$; by definition of the preservation step, this check used the pessimistic labels. Bruns and Godefroid's theorem therefore implies that $F', s_e, R_F \models \psi$ using the regular 3-valued interpretation (neither optimistic nor pessimistic). The truth value of ψ at s_e in P_C depends on the labels copied to the dummy interface states that s_e reaches in F'. If we can argue that those labels remain valid on the actual interface states in P_C, then this case of the soundness proof holds.

  By assumption, R_F strengthens R_P (Definition 12). Lemma 5 implies that R_C strengthens R_P for all propositions in ψ. Based on this relationship between R_C and R_P, Lemma 2 guarantees that for all properties ϕ and states s in P, $P, s, R_P \models_{\mathrm{ST}} \phi \leq P, s, R_C \models \phi$ (*i.e.* that a strengthening model check in P is no more precise than the corresponding regular model check in P). For each state s_c in S_{connect}, $P, s_c, R_C \models \phi$ implies $P_C, s_c, R_C \models \phi$ since the set of states reachable from s_c in P_C is the same as the set of states reachable from P. This establishes this case of the theorem.

  If the preservation algorithm reports ψ as failing based on the check in the optimistic model, the result holds by a similar line of reasoning (replacing uses of pessimistic with optimistic).

- Assume R_F weakens R_P for all evolving propositions in ψ but the optimistic weakened labels do not map to false. Then the proof follows that for the previous case, substituting the corresponding lemmas on weakened interpretations for those on strengthened interpretations.

- In the remaining cases (R_P and R_F are logically equivalent, or no prior case produces concrete results), the proofs are analogous to the proofs for these cases from Theorem 1.

□

6 Case Study Results

Our interfaces are only effective if they enable us to perform most preservation checks compositionally. To evaluate the effectiveness of our interfaces, we searched for feature interaction errors in the email application described in Section 2. We used the case study to determine

- whether our interfaces and methodology can detect the feature interaction errors compositionally,
- the extent to which each aspect of our methodology (original feature-oriented model checking, 3-valued model checking, and evolving propositions) contributed to detecting actual interactions, and
- whether interactions can be detected through combining small numbers of features.

Our experiments use a model checker that we built specifically for handling our feature-oriented verification methodology. We do not present performance figures here in part because because the state machines for these models are too small to generate meaningful performance figures, and because the emphasis in developing the model checker has been to support the methodology rather than provide high performance.

We manually extracted the ten properties described in Section 2 from the interactions that Hall reported in his study [Hal00]. Hall detected twenty-six interactions, of which we detected sixteen.[5] Of Hall's remaining ten interactions, two were too simple to detect at our level of model (we would have had to artificially design a model to reflect the interactions, and the detection would have then been trivial). Two arose from properties that could be expressed in LTL, but not in CTL. Two appeared to require a property specification language that supports alternation. Two interactions involved human concepts such as rudeness that didn't translate well into logical formulas. Finally, two required a remailer with different behavior than the one we had designed based on the remainder of the study.

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.[6] Table 1 summarizes the feature interactions that we

[5] Our tables of results show only fifteen rows because the first of the property 7 entries captures two related interactions from Hall's study.

[6] For the rest of this section, we will implicitly assume that features are composed with the basic mail delivery feature prior to verification; this defines the propositions mail and deliver.

detected using our modeling and verification methodology. Each row describes the property (from Section 2) 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 a statement of which techniques detected the interaction. The values in the table for the last column indicate one of three techniques: the original compositional methodology, 3-valued checks, and strengthened/weakened comparisons.

The tables show several results. First, seven of the sixteen interactions required only the original methodology. The remaining interactions required some combination of the enhancements.

Our methodology detected the five interactions marked with "pessimistic strengthened" based solely on the information in the interfaces; no additional model checking runs were performed during the preservation step. In these cases, the verification step determined that strengthening the evolving propositions would lead to a violation of the property and recorded this fact in the interface. When the violation did occur, the model checker extracted the counter-example already stored in the interface.

We detected two interactions using evolving propositions sans 3-valued model checking; these are marked with "Original" in the techniques column and a non-empty Re-Interpretation column. In these cases, the preservation step required model checking, but only for 2-valued logic. Finding the remaining two interactions required both 3-valued model checking and evolving propositions; in these cases, the information stored about weakening and strengthening was not enough to indicate a violation, so the preservation step ran the 3-valued model checker, using the extended interpretation listed in the table. In no case did we have to verify the full composition of the listed features in order to detect an interaction.

In nine of the sixteen interactions, the propositions evolved at composition time. In all of these cases, the new interpretations always either strictly strengthened or strictly weakened their earlier interpretations; due to our stored interface information in these cases, we never needed to re-verify a property already proven of a feature after re-interpretation. This clearly shows that any methodology for verifying feature-oriented designs must accommodate evolving propositions. The propositions do, fortunately, seem to evolve predictably, which verification techniques should exploit.

The distinction between control and data propositions was necessary to handle four of the interactions, specifically, the ones that violated properties 1 and 4. Each of these compositional checks would have failed if the control propositions had been interpreted as unknown, rather than as false, during model checking.

This case study suggests that our enriched methodology is crucial for detecting many interactions. Our original technique could not find several of the feature interaction problems in this suite. In fact, our original modeling technique could not even model the suite accurately due to the lack of support for evolving propositions. We believe the enriched technique better reflects the modeling and verification needs of a broad range of realistic software systems.

An Interesting Interaction

Property 4, which requires an encrypted message to never be decrypted and then mailed without first being re-encrypted, led to some interesting results during this study. The interactions arising from this property cannot occur with fewer than three features:

- The property holds of the encryption feature alone.
- The property holds when the decryption feature is composed with encryption because the decryption feature does not itself mail anything.
- The property holds when encryption is composed onto either autorespond or forward because the message stays encrypted until mailed.
- The property fails when autorespond or forward is composed with encryption followed by decryption 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 also fails when decryption follows either encryption and autorespond or encryption and 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.

This property differs from the others that yielded undesirable interactions because multiple orders of composition among the features expose the interaction; furthermore, different techniques (3-valued checks versus evolving propositions) exposed the interaction depending upon the composition order.

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 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 [FK01]. 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 [Jac00]) 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 [Var01]. 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 Other Related Work

Compositional verification has a long history dating back to Abadi and Lamport's work on assume-guarantee reasoning [AL95]. In this framework, a designer states manually-developed constraints (assumptions) on the behavior of a module as part of its interface; this framework was designed to support separate development of components. Proof rules govern when a composition of modules is valid according to the assumptions, and dictate when safety properties hold of a composition of modules.

Pnueli [Pnu84], McMillan [McM97], and others have developed proof rules for compositional model checking; these frameworks capture module constraints through temporal logic formulas. These works, however, are really about *decompositional* verification, in which the whole system is available at the same time, but is verified piecewise for tractability. Having the whole system specification enables designers to derive assumptions about the behaviors of the surrounding system. Our modules, in contrast, are developed independently of their eventual deployment context. We

Property	Features Involved	Problem Description	Re-Interpretation	Verification Techniques
1	sign, forward	The sender field of a signed message can be altered by a forwarding feature, and then mailed out.	sender-unchanged strengthened from true to ¬forward	Original
1	sign, remail	The remailer changes the sender field of a signed message.	sender-unchanged strengthened from true to ¬anonymize	Original
2	sign, remail	Signing a messages gives away the identity irrespective of whether the sender field is changed.	anonymous strengthened from anonymize to anonymize ∧¬signed	Pessimistic Strengthened
3	encrypt, verify	If a message is signed and then encrypted, the encryption defeats signature verification.	verifiable strengthened from true to ¬encrypted	Pessimistic Strengthened
4	encrypt, decrypt, forward	A message can be encrypted, mailed out, decrypted, and then forwarded in the clear.	decrypted weakened from false to decrypt-successful	3-valued check
4	encrypt, decrypt, auto-respond	A message can be encrypted, mailed out, decrypted, and then auto-responded such that the auto-response contains the original text of the message.	decrypted weakened from false to decrypt-successful	3-valued check
5	encrypt, remail	A message intended to be remailed cannot be processed by the remailer if the message is originally encrypted.	in-remailer-format strengthened from true to ¬encrypted	Pessimistic Strengthened
6	auto-respond, filter	The filter feature can potentially discard messages generated by the auto-responder.		Original
7	forward, remail	If a user establishes a pseudonym on a remailer and forwards to that pseudonym, then any message sent to the user will be forwarded to the remailer, sent to the user, forwarded to the remailer, etc.		Original
7	forward	A user can provision a forward messages back to himself, thus creating an infinite loop.		Original
7	forward, mailhost	If forwarding is setup to a non-existent user, then the mailhost generates error messages that are then forwarded back to the non-existent user, resulting in longer and longer error responses from the mailhost.		Original
8	forward, filter	The filter feature can potentially discard forwarded messages.		Original
9	auto-respond, decrypt, encrypt	An encrypted message can fail decryption and thus be given to the auto-responder in which it cannot read the subject line.	clear strengthened from true to ¬encrypted.	Pessimistic Strengthened
10	remail, sign	The remailer will alter the body of a signed message if the user wants remailing.	body-unchanged strengthened from true to ¬anonymize.	Pessimistic Strengthened
11	filter, mailhost	If a user sends a message to an unknown recipient at a mailhost, then error messages from that mailhost can be discarded by the filter.		Original

Table 1: Each feature interaction is listed with the property it violates, the interpretation of propositions it requires, and the verification techniques used to expose the problem.

can, nevertheless, exploit the sequential composition in our framework to automatically derive temporal logic interface constraints that must hold at composition time. de Alfaro and Henzinger capture interfaces through automata [dAH01] for parallel composition contexts.

Houdini infers annotations for modular checking in ESC/Java [FL01]. The framework infers candidate annotations through static analysis, then uses ESC/Java to check whether the annotations satisfy the program; if so, the annotations can become part of the program's interface. Our approach differs in several ways. First, we infer properties during individual feature verification. Second, our interfaces capture sufficient information to preserve properties upon composition; Houdini's annotations are not property-driven, and thus may not be useful for a given property. Finally, our approach is truly modular in that we do not require information about the modules we may compose with in order to derive our interfaces; Houdini requires some assumptions on the remainder of the program to perform its modular analysis.

Our case study uses modular model checking to detect certain feature interactions. Feature interaction problems have received substantial attention in the software engineering literature [KK98, Zav97]. Our emphasis here is on modular verification, not on model checking as a tool for detecting feature interaction. Several researchers have attempted the latter in non-modular settings [ABdR00, BA94, JZ98, KK98]. While we appreciate that model checking has limitations in detecting feature interaction, we believe our work enhances the options for using it when applicable in this domain.

Our email example uses a pipe-and-filter model of feature composition; this model resembles Zave and Jackson's Distributed Feature Composition [JZ98]. Our work differs because our full methodology supports features that span multiple actors; we do not cover multiple actors in this paper as they are orthogonal to our discussion of module interfaces. Our work also differs in that its focus is on verification rather than architecture and specification.

Other verification researchers have discussed methodologies for reasoning under sequential composition [AGM00, AY98, CH00, LG98]. These efforts differ from ours in many ways: none handle open systems, none were created towards supporting cross-cutting design methodologies, and all arise in a decompositional verification context rather than a modular design one. Our interfaces and verification methodology are designed to support modularity at the design level.

9 Conclusions

The automated verification of modern software systems requires effort in two directions. First, it must address the structure of modern software: as a third-party composition of independently-produced components that, increasingly, encapsulate software features (as in a product line). Second, it must realize that, even as this style of software could greatly benefit from sophisticated verification techniques, programmers are unwilling and sometimes even unable to write the specifications necessary for verification tools. Automatically synthesizing a suitable alternative to these specifications is a critical software engineering challenge.

The verification technique used in this paper is model checking, restricted to a modular context. Modular verification is critical in this domain for several reasons. Most important of all, there is usually no clear notion of a "whole" program, since independent fragments may be produced by several different developers. In addition, the sizes of whole programs can easily defeat the various techniques model checkers deploy to combat state explosion.

This paper's contributions are twofold. First, it presents a series of definitions of interfaces that support modular verification in this component-based programming universe. The definitions grow to handle both the nature of the software itself, and the needs of the verification methodology. Second, it presents a study of verifying a suite of email features. Our technique identifies most of the feature-interaction problems previously found manually in this case study, thus validating the utility of our interfaces.

This work does suffer from the problem that a feature developer may not know which particular features to verify together to detect errors. This is not a problem for a client, who presumably handles only a particular composition. Even a producer can, however, exploit our methodology to identify potential problems. As Section 6 showed, a failed pessimistic strengthened test stores a counter-example in the interface. Thus, any other feature that strengthens a feature's propositions is guaranteed to raise an error: the developer can detect this *without even verifying the second feature*. This (and, dually, optimistic weakening) should provide a useful diagnostic for a feature developer.

There are numerous directions for future work. Naturally, we need to conduct more case studies to identify other weaknesses in our interfaces. Second, we need experience with a broader user base to determine the true usability of

our tools. More significantly, we intend to explore other kinds of verification tools, such as declarative specification solvers, that might better support the incomplete information that we currently model with 3-valued logic.

References

[AA97] Pansy Au and Joanne M. Atlee. Evaluation of a state-based model of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 153–167. IOS Press, 1997.

[ABdR00] 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.

[AGM00] 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.

[AL95] Martin Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AR98] 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.

[AY98] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Symposium on the Foundations of Software Engineering*, pages 175–188, 1998.

[BA94] K. Braithwaite and J. Atlee. Towards automated detection of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 36–59. IOS Press, 1994.

[BBK95] 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.

[BG99] 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.

[BG00] 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.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[CED01] 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.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGP00] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[CH00] 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.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Symposium on the Foundations of Software Engineering*, pages 109–120, 2001.

[FK01] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 152–163, September 2001.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, 2001.

[GJS96] James Gosling, Bill Joy, and Guy Lewis Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[Hal00] Robert J. Hall. Feature interactions in electronic mail. In *Feature Interactions in Telecommunications Systems*. IOS Press, 2000.

[HJS01] M. Huth, R. Jagadeesan, and D.A. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *European Symposium on Programming*, 2001.

[Jac95] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4), October 1995.

[Jac00] Daniel Jackson. Alloy: a lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, February 2000.

[JZ98] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[KK98] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[KVW98] O. Kupferman, M.Y. 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.

[LG98] Karen Laster and Orna Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[McM97] Ken McMillan. A compositional rule for hardware design refinement. In *International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 24–35. Springer-Verlag, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pnu84] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series*. Springer-Verlag, 1984.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. Available at `http://www.cs.rice.edu/ vardi/papers/index.html`, 2001.

[Zav97] Pamela Zave. Calls considered harmful and other observations: A tutorial on telephony. In Tiziana Margaria, editor, *Second International Workshop on Advanced Intelligent Networks*, 1997.