Interfaces for Modular Feature Verification

Harry C. Li
Brown University, USA
hcli@cs.brown.edu

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

Kathi Fisler
WPI, USA
kfisler@cs.wpi.edu

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 [25]. 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 [23] and Java [14] 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" [22]); 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 [13]. Worse, programmers often 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. This idea fails for numerous reasons. First, not all modules are available at the same place, because they are written by independent authors and assembled (in a componential fashion [27]) 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 [10], the total number of combinations of product line features can exhibit combinatorial

Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)
1527-1366/02 $17.00 © 2002 IEEE

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 [8].

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 manually perform.

The verification technique that this paper defines specifically addresses feature-oriented modules. These modules encapsulate individual program features that cross-cut systems [18] and contain the 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.

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. A companion paper [20] contains the algorithmic details.

This paper also demonstrates the utility of our interfaces through a case study. The case study is based on an analysis of an email system originally conducted by Robert Hall [15]. 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 describes our approach to feature-oriented verification and the interfaces that it engenders.

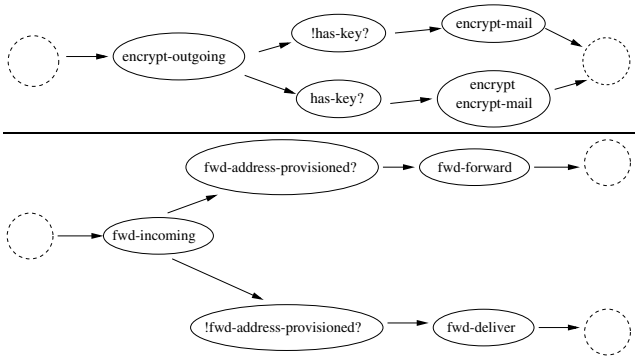

Figure 1. The encryption and forwarding features. Dashed states resolve with concrete ones during composition.

Section 4 presents the results of our case study. Section 5 reviews related work. Section 6 offers concluding remarks.

2 The Email Case Study

The email application offers several features, a characteristic of product line systems; these features can, however, adversely interact with one another in many ways. The application contains 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), mail hosting.

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 as formal properties in the temporal logic CTL [8]. In the descriptions, "deliver" refers to a message that reaches a user on the local mail system and "received" refers to a message that reaches an external recipient.

1. Once a message is signed, the sender field is not altered until the message is delivered or received.
   Formula: $\mathsf{A\,G[\,sign\text{-}msg \rightarrow A[\,sender\text{-}unchanged\ U\ (deliver \vee received)\ ]]}$

2. When a message is ready to be remailed, it is never mailed out with the sender's identity exposed.
   Formula: $\mathsf{A\,G[\,w\ antsRemail \rightarrow A[\,anonymous\ R\ \neg mail]]}$

3. If one tries to verify a signature, then the message must be verifiable.
   Formula: A G[try-verify → verifiable]

4. When a message is encrypted, it is never decrypted and then sent in the clear.
   Formula: A G[encrypt → A[(deliver ∨ received) R A G¬(decrypted ∧ E[¬encrypted U mail])]]

5. If a message is to be remailed, it is formatted correctly for the remailer to process it.
   Formula: A G[toRemailer→ in-remailer-format]

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

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

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

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

10. If an outgoing message is signed, then its body is never changed unless is it delivered or retrieved.
    Formula: A G[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: A G[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 4 describes these interactions and the specific aspects of our methodology that detect the failures.

3 Modular Feature Verification

Intuitively, a feature-oriented module contains the code fragments that implement a feature across the actors of a system. In a full email application, for example, a message-forwarding feature would involve the mail client, the database of user information (to retrieve the forwarding address), and the router that dispatches mail to users. Rather than insert separate code into each of the database, router, and client, feature-oriented programming keeps the code together in the design. This organization makes it easier to add and remove features, thus making feature-oriented programming well-suited to product-line development.

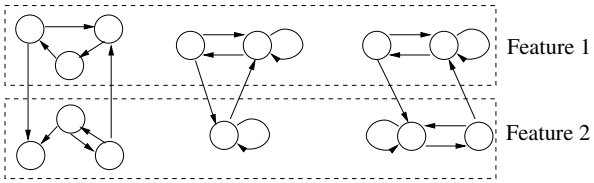

Figure 2. Features and their composition.

Inferring properties about feature-oriented systems from individual features requires a formal semantics of feature composition. Some popular versions of cross-cutting, such as aspects [18], lack the level of formal semantics necessary to support this task. Our work uses a somewhat restricted model of features for which the composition semantics is straightforward. Our model is similar at the high-level to those of Batory [5] and Ossher and Tarr [24], but differs at the low-level. We view features as fragments of state machines and composition as inserting edges between the fragments for the same actor; Figure 2 shows an example. While a state-based model does not completely capture the behavior of the software, it is rich enough to uncover interesting technical problems with compositional verification.

The contents of feature interfaces arise from various aspects of our verification methodology. To simplify the presentation, we describe interfaces in stages, refining them as we cover the methodology in more detail. We begin with our formal model of features and their composition.

Definition 1 A state machine $M = (S, \Sigma, \Delta, s_0, R, T, F)$ is a tuple where S is a set of states, Σ and Δ are sets of input and output atomic propositions, $s_0 \in S$ is the initial state, $R \subseteq S \times \text{PL}(\Sigma) \times S$ is the transition relation, $T : S \rightarrow 2^{\Delta}$ indicates which propositions are true in each state, and $F : S \rightarrow 2^{\Delta}$ indicates which propositions are false in each state $(\forall s \in S, T(s) \cap F(s) = \emptyset)$.[1]

Definition 2 A *feature* is a tuple $\langle E_1, \ldots, E_n \rangle$ of state machines.

Given two features and the states at which to connect the state machine fragments, we can compose the features by inserting edges between the fragments. This leads to our first definition of feature interfaces, as well as a definition of composition:

Definition 3 (*Interfaces, version 1*) An interface for feature $\langle E_1, \ldots, E_n \rangle$ contains the following information:

[1] Our motivation for specifying both true and false propositions in state machines will become clear shortly.

- For each E_i, a set *exit* of states in E_1 from which control can enter another feature.
- For each E_i, a state *re-enter* in E_1 to which control can return from another feature.

Definition 4 Let F_1 and F_2 be features with the same number of component state machines. Let I_1 and I_2 be interfaces of F_1 and F_2, respectively. Composing F_1 and F_2 via I_1 and I_2 yields a tuple of state machines $\langle C_1, \ldots, C_k \rangle$. Each C_i combines the i^{th} machines of F_1 and F_2 by inserting a transition from each state in *exit* of F_{1_i} to *re-enter* of F_{2_i} and from each state in *exit* of F_{2_i} to *re-enter* of F_{1_i}.

3.1 The Core Verification Methodology

Our verification methodology supports 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 a feature that are sufficient to preserve each property after composition.
3. Proving that a feature F_1 satisfies the preservation constraints of another feature F_2 (the *preservation step*). We establish preservation by analyzing at most F_1, not the composition of F_1 and F_2.

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 that adding the new feature will preserve all of these labels. We therefore store these labels in the interface and confirm that they still hold during preservation checks. We refer to this as the *original* methodology later in the paper; the formal details appear elsewhere [12].

The original methodology handled reasoning about control properties, but not data attributes. To handle data in features, our methodology must support reasoning about features as open systems and propositions whose interpretations evolve upon composition; this also compels us to enrich the interfaces. The remainder of this section uses the email case study to illustrate why these needs arise, how they affect our interfaces, and how we use the interface information. The algorithmic details appear elsewhere [20].

3.2 Deriving the Interfaces: The Verification Step

The verification step has two purposes: first, it verifies a property against an individual feature; second, it computes the state labels that we will store in the feature's interface for preservation checks at composition time. Two aspects of our methodology, features as open systems and evolving propositions, affect how we perform this step.

3.2.1 Open Systems

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 [8] will not be able to perform this check, 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.

We handle open systems using Bruns and Godefroid 3-valued CTL model checking algorithm [7]. This algorithm allows propositions to have values true, false, or unknown. We interpret propositions from other features as unknown; the true and false labelings in Definition 1 capture the three values (propositions not labeled with either true or false in a state are interpreted as unknown). 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. In the context of compositional verification, a result of unknown during a preservation check would require us to verify the composition of the features, rather than the individual features. To potentially increase the number of cases yielding concrete results, Bruns and Godefroid perform two verifications which they call *optimistic* and *pessimistic*; in the former, all unknowns are interpreted as true, while the latter interprets them as false. Any property which evaluates to false in the optimistic model is guaranteed to be false, while any property which evaluates to true in the pessimistic model is guaranteed to be true [7]. Thus, a 3-valued model check actually involves two runs of the model checker; we must store the labelings that arise from both runs in our interfaces.

The open systems arising from features differ from those arising from conventional model checking. In conventional model checking, modules compose in parallel; propositions defined in other, abstracted, modules can change value anytime. Features, in contrast, compose sequentially[2]; propo-

[2]Parallel composition occurs *within* features (to synchronize actors), but not across them. Our work exploits this refined architecture [12].

sitions arising from one feature do not change value while another feature is executing. Furthermore, we can partition propositions from other features into *data propositions*, which represent attributes of shared data (such as whether a message is encrypted), and *control propositions* which model the external (user) choices that drive the feature (such as w antsRemailfrom property 2). Control propositions of one feature are never true in another feature because features do not execute simultaneously. Thus, we can set the control propositions from other features to false (rather than unknown) during model checking; reducing the number of unknowns increases the likelihood of obtaining concrete results during model checking.

Definition 5 (*Interfaces, version 2*) An interface for feature $\langle E_1, \ldots, E_n \rangle$ contains the following information:

- For each E_i, a set *exit* of states in E_1 from which control can enter another feature.
- For each E_i, a state *re-enter* in E_1 to which control can return from another feature.
- A partition of E_i's propositions ($\Sigma \cup \Delta$) into control and data propositions.
- For each state q in *exit* $\cup \{$*re-enter*$\}$, two sets of CTL formulas: one containing the labels ascribed to q during the pessimistic check, and the other containing the labels ascribed to q during the optimistic check.

3.2.2 Evolving Propositions

Consider property 2, which requires messages passed through the anonymizing remailer to not reveal any information that identifies the sender. What is the definition of anonymous in this property? From the perspective of the remailer feature alone, anonymous is the same as the proposition remail-anonymize assigned in the remailer. Once we add the signing feature, however, a message also needs to be unsigned in order to be considered anonymous.

This example illustrates how adding features may change our interpretation of existing propositions. In this sense, the propositions in our formulas evolve over time; in our running example, they capture concepts that arise in email systems, but the concrete definitions of those concepts may change based on the features included in the model. Our interfaces must therefore handle evolving propositions while retaining compositionality.

Our methodology views evolving propositions as propositions that do not label states in the state machines but that may appear in properties. As we compose features, we resolve the evolving propositions into boolean combinations of concrete propositions from the features. The following definition captures these bindings. We model check a formula under an interpretation by replacing all evolving propositions in the formula with their bindings under the interpretation.

Definition 6 An *interpretation* is a function from evolving propositions to boolean formulas (containing $\wedge$, $\vee$, $\neg$). We assume that no evolving propositions appear in the range of an interpretation.

Verifying properties of individual features requires 3-valued model checking. We wish to increase the number of cases in which future preservation checks can be performed compositionally; as modifying the interpretation of an evolving proposition could affect whether a property holds, we must anticipate the impact of changes to the evolving propositions. Our methodology handles two specific changes: those that logically strengthen a proposition's interpretation, and those that logically weaken it. The methodology thus verifies properties under three interpretations of evolving propositions, as detailed in the following algorithm.

Verification step: Let F_1 be a feature, φ be a property to verify against F_1, and R be an interpretation of the evolving propositions in φ. We verify φ under three interpretations:

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

Failure of the strengthened/weakened checks indicate cases where the property would not hold if the propositions evolved a particular way. In these cases, if the property fails, we also store the generated counterexample in the interface, so we can present it to the user at composition time. The preservation step (Section 3.3) will choose which set of labels to use based on the context of the composition.

Our use of separate strengthening and weakening checks is analogous to Bruns and Godefroid's use of pessimistic and optimistic interpretations, in that it treats the extremal cases. Treating all possible strengthening/weakening cases would result in a combinatorial blowup, which would render the methodology impractical. Our case study demonstrates that properties often involve only one evolving proposition (since properties generally correspond to single requirements), so this approach appears useful in practice.

The combination of evolving propositions and 3-valued model checking leads to the following, final, definition of interfaces:

Definition 7 (*Interfaces, version 3*) The new interface definition replaces the last item from version 2 (Defn 5) with the following information:

- For each state in *exit* $\cup\{$*re-enter*$\}$, six sets of CTL formulas, arising from the pessimistic, optimistic, pessimistic strengthened, pessimistic weakened, optimistic strengthened, and optimistic weakened checks. We store two pieces of information with each set: the interpretation of evolving propositions that was in effect when the formulas were derived, and whether the conditions for that set guarantee the property to hold or be violated. If a property is violated, we also store the counterexample arising from the violation.

The number of different sets of labels required to support compositional preservation checks would overwhelm designers if we asked them to develop the interfaces manually. Fortunately, the CTL model checking algorithm generates the formula sets automatically (model checkers based on LTL would not easily support this task). Thus, while these interfaces are more complicated than standard, type-based interfaces, our ability to generate them automatically makes them tractable for designers to use. As the formulas in these sets are short, representable as strings, and assigned to only a few states, the sizes of our interfaces should not be prohibitive in practice.

3.3 Using the Interfaces: The Preservation Step

We use the preservation step to check that composing features F_1 and F_2 will preserve a property φ already proven of F_1. In general, we perform this step by attaching two dummy states to F_2, representing the interface states of F_1 to which F_2 will attach. We then seed the dummy state that exits F_2 back into F_1 with the labels assigned to the actual re-entry state from F_1; these labels are stored in F_1's interface. We then use the CTL model checking algorithm to check that each label from F_1's exit interface state holds in the dummy state that enters F_2 [12].[3] If all of these labels are preserved, φ is guaranteed to hold in the composition. Because propositions evolve, however, a given composition does not need to preserve all of the labels in the interface; the following algorithm indicates which ones are relevant.

Preservation step: Let R_1 be the interpretation used to verify φ in F_1 and let R_2 be the new interpretation associated with F_2. We perform the following sequence of checks to determine whether φ is preserved in the composition of F_1 and F_2 under R_2:

- If $R_2(p)$ strengthens $R_1(p)$ for all evolving propositions p in φ, check whether the pessimistic strengthened case held in F_1. If so, copy/confirm the pessimistic interface labels that arose under the strengthened interpretation on F_2's dummy states. If not, there is no need to proceed with verification of F_2 because F_1 already violates the property.
- If $R_2(p)$ weakens $R_1(p)$ for all evolving propositions p in φ, follow the previous case using pessimistic weakened in place of pessimistic strengthened.
- If $R_2(p)$ is logically equivalent to $R_1(p)$ for all evolving propositions p in φ, copy/confirm the pessimistic interface labels that arose under R_1 (with no strengthening or weakening) on F_2's dummy states.
- In all other cases, re-verify φ against F_1 using R_2, then apply version 2 of the preservation algorithm to check preservation in F_2.

4 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 [15]. Hall detected twenty-six interactions, of which we detected sixteen.[4] 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

[3] CTL "until" properties also give rise to additional checks, but the formulas to check are also derived automatically and stored in the interface.

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

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

[5] 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.

among the features expose the interaction; furthermore, different techniques (3-valued checks versus evolving propositions) exposed the interaction depending upon the composition order.

5 Related Work

Compositional verification has a long history dating back to Abadi and Lamport's work on assume-guarantee reasoning [1]. 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 [26], McMillan [21], 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 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 [11] for parallel composition contexts.

Houdini infers annotations for modular checking in ESC/Java [13]. 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 [17, 28]. 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 [4, 6, 16, 17]. 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 [16]. 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 [2, 3, 9, 19]. 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.

6 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,

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 forwared 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.	clea rstrengthened 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.

who presumably handles only a particular composition. Even a producer can, however, exploit our methodology to identify potential problems. As Section 4 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.

Acknowledgements: We thank Bob Hall for discussions about his case study, Colin Blundell for his feedback, and the anonymous reviewers for their detailed comments.

References

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

[2] 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.

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

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

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

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

[7] 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.

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

[9] 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.

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

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

[12] K. Fisler and S. 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, Sept. 2001.

[13] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, 2001.

[14] J. Gosling, B. Joy, and G. L. Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.

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

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

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

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

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

[20] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *Symposium on the Foundations of Software Engineering*, 2002.

[21] K. 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.

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

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

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

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

[26] A. 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.

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

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