Parameterized Interfaces for Open System Verification of Product Lines*

Colin Blundell
University of Pennsylvania

Kathi Fisler
WPI

Shriram Krishnamurthi
Brown University

Pascal Van Hentenryck
Brown University

Abstract

Software product-lines view systems as compositions of features. Each component corresponds to an individual feature, and a composition of features yields a product. Feature-oriented verification must be able to analyze individual features and to compose the results into results on products. Since features interact through shared data, verifying individual features entails open system verification concerns. To verify temporal properties, features must be open to both propositional and temporal information from the remainder of the composed product. This paper addresses both forms of openness through a two-phase technique. The first phase analyzes individual features and generates sufficient constraints for property preservation. The second phase discharges the constraints upon composition of features into a product. We present the technique as well as the results of a case study on an email protocol suite.

1. Introduction

Feature-oriented architectures organize code around the features that a system contains [33]. By better aligning the implementation of a system with the external view of users, feature-orientation offers several potential benefits for software engineering such as ease of maintenance, evolution, and verification. As a result, this style of organization is at the heart of increasingly important development methodologies such as product-line software [16], and provides a meaningful framework for component reuse.

There is growing support for *development* around features (and the related, more general notion of aspects) [4, 5, 29, 30, 33], but this work largely ignores key questions of formal verification. In principle, feature-orientation can simplify verification because both features and requirements arise from a user's view of a system [17]. We cannot employ a brute-force approach to verification by constructing each product individually and verifying it, because there is a combinatorial number of products in the number of features, and verification is expensive as individual products grow larger. Ideally, therefore, we would like to verify requirements against individual feature modules, then perform *lightweight* checks to ensure that composition does not violate these properties. This is challenging because features interact subtly, often through shared state [8, 24].

A modular form of feature verification must support data that propagate across features. This in turn depends on techniques for handling propositions whose values may not be available when analyzing a single feature because they are defined elsewhere, making features a form of *open system*. This paper presents a verification technique, inspired by model-checking and applying ideas from flow analysis, that generates constraints on individual features, then discharges the constraints during composition to establish system-wide properties. The novelty of this technique is that the constraints are parameterized over the information that makes features open; furthermore, they are parameterized *differently* based on the nature of the open information. This results in generated interfaces that are more concise and precise than in previous modular feature verification techniques. Our new technique also lifts properties of individual features to composed systems, whereas prior techniques checked only for local interactions between features.

One observation from our work is that *traditional model checking does not mesh well with the needs of modular verification of features*. Model checking is primarily designed to authoritatively determine the truth or falsity of properties over models. However, most of the property violations we observe arise only upon composition, because some compositions satisfy properties while others fail them. Therefore, most verification runs over individual features are (and must be) inconclusive, pushing the burden onto the composition step. Trying to map this to a traditional model checking framework can be unsatisfying due to the preponderance of inconclusive answers (see Section 3.2). Instead, the problem becomes one of generating constraints as interfaces to be discharged during composition, rather than merely checking properties. While interface generation in general is hard, we use properties to make this process tractable, resulting in a technique that employs *property-driven interface generation*.

* Partially supported by NSF grants CCR-0132659, CCR-0305834 and CCR-0305950.

The key parts of the paper are an overview of the subtleties of modular feature verification and an outline of our prior approach to this problem (Section 3), our new technique (Section 4), experimental results (Section 5), and a discussion of perspective and limitations of our result, including directions for future work (Section 6).

2. Background on CTL and Model Checking

Model-checking [15] is an automated verification technique used to establish properties of finite-state systems. A model-checker consumes a description of a system, usually given as a state machine, and a specification of a property that the system must obey. The state machine can be non-deterministic. The property is typically written in a temporal logic such as CTL [15].

The atoms of CTL are propositions that label states. CTL permits combination of these atoms using the standard propositional operators and connectives (negation, conjunction, implication, etc). In addition, CTL can capture *temporal* properties. A formula of the form $[\phi \ \mathsf{U} \ \psi]$ (where ϕ and ψ are both CTL formulas) is true at a state if ϕ is true now and in the future until a state where ψ is true (read the U as "until"). Because many paths leave a state, we must quantify this formula by whether we expect the property to hold in all possible future worlds or only in some. The CTL formula $\mathsf{A}[\phi \ \mathsf{U} \ \psi]$ expects that on All paths, ϕ will hold in every state until a state where ψ is true, while $\mathsf{E}[\phi \ \mathsf{U} \ \psi]$ requires that there Exists a path where this holds. In this paper we also use the unary operators AG, whose sub-formula must hold in all states, and EF, whose sub-formula must hold in at least one future state.

3. Problem Motivation

3.1. Why is Feature Verification Subtle?

Consider an email feature suite that includes components for anonymous remailing and message signing.[1] A product might include these two features and basic mail delivery, as shown in Figure 1. In this paper we employ state machines to model systems, relying on either program analyses over source, or state machine domain-specific languages, as sources of these models.

A feature consists of both a state machine and a set of interfaces. An interface specifies states to which new features attach (via both incoming and outgoing edges). In the REMAIL feature from Figure 1, for example, the interface would specify that edges leave from states r_1 and r_2 and enter at r_0. Features within a system compose in a pipe-and-filter architecture [32], beginning and ending in some basic infrastructure that is common to all products within the family (such as basic mail delivery, in the email example); we call this the *base product*. Composing features into products involves adding edges between interface states.

In our running example, the requirements state that when a product uses the REMAIL feature, a message marked for anonymous remailing should remain anonymous until it is mailed. For this set of components, anonymity means a message has been marked for remailing and not been digitally signed. The temporal logic formula

$$\phi = \mathsf{AG}(\mathit{remailed} \rightarrow \mathsf{A}[(\mathit{remailed} \wedge \neg \mathit{signed}) \ \mathsf{U} \ \mathit{mailed}])$$

captures this property. The sample product in Figure 1 would violate this property on a path that includes the upper path of states through the SIGNING feature. The compositional verification challenge is to detect this interaction without model checking the full product, because the total number of products explodes combinatorially in the number of features, making it infeasible to conduct an expensive whole-program traversal over each product. Instead, we must analyze the features separately, generate appropriate interfaces on each feature for preserving properties, and check for interactions by combining interface information at product assembly time.

The CTL model-checking algorithm does not inherently support modular feature verification. For instance, if a model checker evaluated ϕ in the initial state of the REMAIL feature alone, it would report the property as false because REMAIL does not mention the proposition *mailed* (therefore assumed to be false, which violates the AU). However, when verifying individual features, a model checker cannot assume that propositions are false simply because they do not label states of a feature: some propositions are asserted in later features, while others are asserted prior to executing a feature and their values persist until explicitly changed.

Consequently, any verification algorithm must distinguish between two different uses of propositions of unknown value within a state machine, which we name *control* propositions and *data* propositions [28]. Control propositions capture settings from the (user) environment of the system, such as *wantsRemail?* in Figure 1. Data propositions capture attributes of data in the system, such as *signed* and *remailed*. The verifier must treat data propositions as *persistent*: their values hold across features until changed by an assignment. Control propositions are not persistent, getting their value solely from the labelling functions (and assumed false at a state if not explicitly labeled). Our model (definition 3) asks the designer to explicitly identify the data propositions of a feature.

An additional problem is that individual features contain only a portion of the entire product's state space: the model checker therefore lacks information about the prop-

1 These examples come from a suite due to Robert Hall [22].

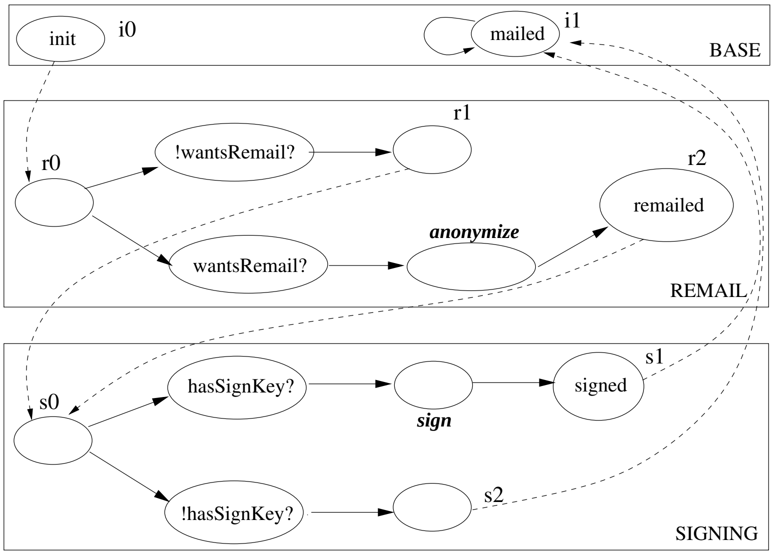

Figure 1. A simple email product with remailing and message signing. The dashed transitions show one possible assembly of features into a composed product. Another assembly might permute the order of features. In the state machines, ! denotes logical negation, propositions ending in ? represent control decisions, and all other propositions represent data attributes of email messages. Identifiers next to states name those states for reference throughout the paper, except *anonymize* and *sign*, which are abstractions representing portions of the state machine whose details are not relevant to this example.

erties that hold along paths that emanate from the feature. For REMAIL to satisfy ϕ, paths that leave from state r_2 must eventually mail the message; it is possible to use REMAIL in two different products such that one satisfies this property and one does not. At best, therefore, a model checker can only traverse the feature and determine what constraint it imposes on the features that eventually connect to it.

3.2. A Problematic Prior Approach

These scenarios point to two problems in the use of model checking for modular feature verification: traditional model checkers are based on binary logic and on closed-world assumptions. Three-valued model checking [9] appears to address both problems. In prior research [28], we exploited three-valued logic for modular feature verification by setting all unknown propositions in a feature to $\perp$ prior to model checking. However, checking property $\psi = \mathsf{AG}(\mathit{remailed} \rightarrow \mathsf{A}[\neg \mathit{signed}\ \mathsf{U}\ \mathit{mailed}])$ against the SIGNING feature in this framework would return the value $\perp$, because under this substitution the property reduces to $\mathsf{AG}(\perp \rightarrow \mathsf{A}(\neg \mathit{signed}\ \mathsf{U}\ \perp))$. This result indicates nothing other than that the truth of the property depends on more than the variables in the SIGNING feature. Worse still, reducing the unknown variables to $\perp$ before verification prevents reasoning about the property once features are composed and actual values for those variables are available. Used naively, this approach seems worse than no modular checking at all!

To ameliorate this situation, our prior work analyzed features relative to certain specific combinations of values for unknown propositions (as described in Section 5). This unfortunately leads to a potentially exponential number of both model checking runs and interfaces for modular verification. A more scalable solution is clearly required.

4. The Solution

4.1. Insight

Mapping all unknown propositions to $\perp$ early in modular verification incurs potentially high overhead in order to allow specializing interface constraints with values of unknown propositions when they become known (at composition time). The key insight of this paper is: *parameterizing interfaces over the unknown propositions controls interface explosion without sacrificing precision*. A secondary, and more subtle, insight is that *we can parameterize over these propositions differently depending on whether they are defined in prior or subsequent features.* From the perspective of the SIGNING feature on property ψ, for example, the *remailed* proposition is constrained propositionally (based on its value from REMAIL), while *mailed* is constrained temporally (based on paths that satisfy it through MAIL).

Given a feature F and a property, our methodology generates a constraint consisting of two formulas, one propositional and the other temporal. The propositional formula summarizes the effect of F on data propositions. The temporal formula constrains the behavior of features that follow F; since the validity of the property in F may depend on values of propositions set in prior features, this temporal formula is parametric over the unknown propositions of F. (This association of the temporal constraint with features that follow F arises from our use of CTL, which is a future-time temporal logic.) These constraints become part of a feature's interface.

We use the interfaces generated for each feature and property to determine whether that property holds over a composition of features. This reduces to the problem of instantiating the parameterized temporal constraints and checking their validity. The values for the parameters come from both the propositional summaries of prior features and the validity of temporal constraints of later features.

The rest of this section uses the email example to illustrate our technique in more detail. We will assume that a product family will be built from a set of n features and a base product for the family. Assume that the requirements for features in the family are known, and have been expressed as CTL formulas.

4.2. Models of Features and Products

Space constraints limit us to a partial explanation of the foundations backing our informal presentation. We request the interested reader to consult our extensive technical report [7], which offers a complete formalization as well as full proofs of soundness.

Our formal model of feature-oriented systems views each feature as a single state machine with potentially many initial states. In realistic systems, many entities participate in a feature, so a feature would be defined by a parallel composition of state machines for each such entity. Our previous work shows how to reduce models where each feature has multiple state machines to the single-machine model [17], so we adopt the single-machine model here for simplicity.

Definition 1 A *state machine* is a tuple $\langle S, \Sigma, \Delta, S_0, R, Tr, Fa\rangle$ where

- S is a set of states,
- Σ and Δ are sets of input and output propositions,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times PL(\Sigma) \times S$ is the transition relation, where $PL(\phi)$ denotes the set of propositional logic expressions over the set of propositions in ϕ,
- $Tr : S \to 2^{\Delta}$ indicates which propositions are set to true in each state, and $Fa : S \to 2^{\Delta}$ indicates which propositions are set to false in each state ($\forall s \in S, Tr(s) \cap Fa(s) = \emptyset$).

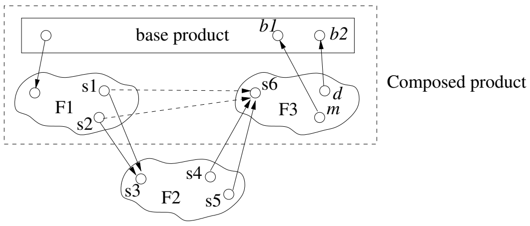

Figure 2. Inserting a feature into a product

This definition is standard, with one important exception. In the style of open systems we analyze here, the law of the excluded middle does not hold: the absence of a label does not imply its falsity. Our model therefore employs distinct labeling functions for true and false labels.

Feature composition adds edges between interface states. Outgoing states have a *connection specification*, a boolean expression over output propositions. Composition adds edges from each outgoing state to all the incoming states whose true and false labels satisfy the connection specification.

Definition 2 A *base product* consists of a state machine M and an interface $\langle\{s_{\text{outgoing}}\}, S_{\text{connect}}, R_{\text{out}}\rangle$ such that if S and R are the states and transitions of M, then $s_{\text{outgoing}} \in S$, $S_{\text{connect}} \subset S$, R_{out} is a set of connection specifications for s_{outgoing}, and R contains edges from s_{outgoing} to each state in S_{connect} (composition replaces these edges with features).

Definition 3 A *feature* is a state machine captured by the tuple $(S, \Sigma, \Delta, S_0, R, Tr, Fa)$ with an interface $\langle S_0, S_{\text{exit}}, R_{\text{exit}}\rangle$ and a set of *data propositions* D where

- All propositions in D lie in the domain of at least one of Tr or Fa.
- $S_{\text{exit}} \subseteq S$ is the set of terminal states of the feature; these states must have out-degree 0.
- R_{exit} is a set of connection specifications for states in S_{exit}.

Our formal model supports two techniques for combining features: features can be composed into compound features, and features (atomic or compound) can be combined with base products to form complete products. Figure 2 shows a product consisting of a base product and two features F_1 and F_3, and the insertion of feature F_2 into this product. The insertion 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 (so control routes through the new feature, with edges replaced by paths) and adds the four edges that connect F_2 to the product.

4.3. Generating Constraints

Given a feature and a property (expressed as a CTL formula), the algorithm generates a *temporal constraint* that is parameterized over the features attached fore and aft, and a *data environment*, which summarizes the persistent data values that the feature passes to subsequent features. The temporal constraints and data environment form a feature's interface. For a given set of properties and features, this interface can be generated once for each feature and reused over multiple product assemblies. We now explain each of these in turn.

The temporal constraint generator is a variant of the traditional CTL algorithm. It runs on every subformula of every property and, instead of returning only truth or falsehood, generates a formula in a variant of CTL. This variant language is easily explained with an example. Consider the property

$$\varphi = \mathsf{AG}(\mathit{remailed} \rightarrow \mathsf{EF}\mathit{mailed})$$

which states that a message marked as remailed can eventually be mailed. Generating constraints on φ at the initial state (r_0) of REMAIL yields the annotated formula

$$\begin{array}{ll} & \varphi_{r_1} \wedge (\neg \mathit{remailed} \vee (\mathit{remailed} \wedge (\mathsf{EF}\mathit{mailed})_{r_1})) \\ \wedge & \varphi_{r_2} \wedge (\mathsf{EF}\ \mathit{mailed})_{r_2} \end{array}$$

The subscripts (*tags*) on formulas contain names of terminal states in the feature; a tagged formula denotes the value of that formula in the successor states to the tag state (these values are available at product assembly time). Intuitively, this constraint says that the entire property must hold in the successors to both r_1 and r_2 (from φ_{r_1} and φ_{r_2}): this is expected, since an AG property must hold in every state of the composed system. The constraint further requires that control can eventually reach a *mailed* state from r_1 unless *remailed* is already false. The constraint is simpler for r_2, because a path to r_2 is known to satisfy the *remailed* proposition. In general, the generated formula uses names of propositions to parameterize over the data environment and tagged subformulas to parameterize over the exit paths.

In contrast to the temporal constraint, which *delimits* the behavior of subsequent features, the data environment *provides* persistent propositional values to subsequent features. Each feature effectively updates the data environment in turn for the next feature in the product.

A data environment maps each terminal state s_t in a feature to a set of of pairs $\langle s_0, V \rangle$, where s_0 is an initial state and V captures the latest values assigned to propositions along a path from s_0 to s_t. Informally, the data environment for the REMAIL feature shown in Figure 1 maps state r_1 to

$$\langle r_0, \emptyset \rangle$$

(because no data propositions occur on a path from r_0 to r_1) and maps state r_2 to

$$\langle r_0, \{\langle \mathit{remailed}, \mathsf{true} \rangle\} \rangle$$

The proposition *wantsRemail?* does not appear in the data environment because it is a control proposition. Similarly, the SIGNING feature's data environment maps s_1 to $\langle s_0, \{\langle \mathit{signed}, \mathsf{true} \rangle\} \rangle$ and state s_2 to $\langle s_0, \emptyset \rangle$. Data propositions are initialized to false.[2]

Formal Details The details in this section are intended for readers already familiar with the CTL model checking algorithm. Due to space constraints, we defer the full algorithms to the technical report.

Intuitively, the constraint-generation algorithm (henceforth called CONSTRAIN) partially evaluates the given property over the feature, under the assumption that data propositions persist along paths. The algorithm handles persistence by storing the most recent value of each data proposition along the current path in a variable (*path-env*).

CONSTRAIN has the same recursive structure as the CTL model checker, but diverges from it to parameterize over data environments and terminal state properties during this partial evaluation. The significant deviations are in the treatment of propositions and of terminal states. For propositions, the result of the CONSTRAIN algorithm depends on the nature of the proposition:

- The values of data propositions of the feature being verified come from the *path-env* argument.
- If the proposition is a control proposition of the feature, its value comes from the labeling functions *Tr* and *Fa*. Control propositions capture user or environmental decisions, and do not appear in *path-env*.
- If the proposition is a control proposition of another feature, its value must be false in this feature because control decisions in one feature are inactive in other features. This situation can arise when checking a property that is primarily about one feature in another feature. For example, the *wantsRemail*? proposition in the REMAIL feature should be treated as false while analyzing SIGNING. We provide other examples of such propositions in our prior work [28].
- Otherwise, the proposition is a data proposition of another feature and its value will (eventually) come from the incoming data environment. The CONSTRAIN algorithm inserts the proposition itself into the constraint formula, which parameterizes the constraint over the value from the data environment.

2 This reflects the fact that data propositions usually reflect attributes that are initially not set, but the theory holds if the values are initialized to $\perp$.

In addition, when the CONSTRAIN algorithm reaches a terminal state of the feature, it cannot evaluate the formula as the successor states will not be available until composition time. The algorithm parameterizes the constraint over the possible successors by tagging the subformula that must hold at those successors.

Data environments summarize the values assigned to propositions along paths between specific initial and terminal states of a feature.

Definition 4 *Let F be a feature with data propositions DP. Let Π be a path $s_0, \ldots, s_t$ in F where s_0 is an initial state and s_t a terminal state in F.*

1. *The* data value *for Π is the set of tuples $\langle p, v\rangle$ where $p \in DP$ and v is the last value for p set on Π.*
2. *Let s_t be a terminal state of F. The* data environment of F at s_t *is the set $\{\langle s_0, DV\rangle\}$ such that s_0 is an initial state with a path to s_t and DV is the set of data values for all paths from s_0 to s_t.*[3]

The constraint discharge algorithm will use data environments to look up the last value given to a proposition along paths between a particular initial and terminal state.

Subtlety When computing the data environment for the base product, the algorithm first removes all edges between the interface states. Base products generally contain edges that restart the product on new data (such as an edge from the *mailed* state to the *init* state in Figure 1, not shown in the figure). These edges can cause data propositions to incorrectly leak across runs of the product. Removing these edges ensures that the data environment of the base product accurately reflects the data available to the features at the start of each new pass through the product.

4.4. Discharging Constraints

Discharging a constraint entails reducing it to a concrete logical value at composition time. Recall the constraint generated from property φ for REMAIL:

$$\begin{array}{ll} & \varphi_{r_1} \wedge (\neg remailed \vee (remailed \wedge (\mathsf{EF} mailed)_{r_1})) \\ \wedge & \varphi_{r_2} \wedge (\mathsf{EF}\ mailed)_{r_2} \end{array}$$

To reduce this to a value, we need concrete values for the *remailed* proposition and for the tagged temporal formulas. The value of the former comes from the data environment summarizing the preceding features. The values of the latter come from discharging constraints on the subsequent features, and propagating the results. The process involves three steps.

3 A standard fixpoint construction handles infinitely many paths.

First, the constraint generation phase produced data environments for each preceding feature individually. To summarize the preceding features collectively, we compose their data environments; this composed data environment offers a concrete logical value for each proposition in the constraint to be discharged. Composing data environments D_i and D_j yields a data environment over the propositions in either D_i or D_j, where values from D_j override values for the same propositions from D_i. Composing the data environments given above for REMAIL and SIGNING (corresponding to feature composition via the dashed transitions in Figure 1) yields this data environment at s_1:

$$\{\langle s_0, \{\{\langle signed, \mathsf{true}\rangle, \langle remailed, \mathsf{true}\rangle\}\}\rangle\}$$

and correspondingly at s_2: $\{\langle s_0, \{\{\langle remailed, \mathsf{true}\rangle\}\}\rangle\}$. Composing data environments for two features potentially yields one data value for each pair of data values in the features being composed. (While this can explode the size of the data environments, we can always use a meet-over-all-paths analysis [25] and, when paths assign conflicting values to a proposition, set the value of the proposition to $\top$.)

Second, we substitute propositions with their values from the composed data environment. In the constraint on REMAIL, for instance, we replace *remailed* with the value true from the data environment above.

Finally, the values of the tagged temporal formulas come from the results of discharging constraints on the subsequent features. Assume we have finished discharging constraints on the SIGNING feature, which follows REMAIL in our running example. Discharging constraints on SIGNING results in concrete values for each subformula of φ in the initial state s_0 of SIGNING. The constraint on REMAIL contains the tagged subformula $(\mathsf{EF}\ mailed)_{r_1}$. We substitute the value of $(\mathsf{EF}\ mailed)$ in s_0 for $(\mathsf{EF}\ mailed)_{r_1}$ (because s_0 is the successor to r_1 in the composed system).

Repeating this step for each tagged subformula in the constraint yields a propositional formula, which a validity check reduces to a concrete value. If evaluating a subformula under two different data values yields conflicting propositional values, we set the result to $\top$. In any case, the result becomes the value of φ_{r_0} when discharging constraints on the preceding features. Note that this step requires only substitution of previously computed results, not model checking; hence the approach is compositional. This step may, however, require propositional reasoning in the presence of $\top$ values for some tagged subformulas (hence rendering the method incomplete).[4]

4 This method is unoptimized and checks the values of many constraints whose values are unused at assembly time. An optimized version would check only those constraints that are needed to discharge the tagged formulas in other states.

Summary To summarize the process, assume the client has chosen a sequence of m of the original n features to assemble into a product, and has composed the features in order along with a base feature. Let $F_1,\ldots,F_m$ denote features, D_i the data environment induced by F_i, $\circ$ a composition operator for data environments, C_i the temporal constraint on F_i, and check(C_i) the result of discharging constraint C_i. The following steps summarize the methodology:

step	*compute*	*using*
1.	check(C_m)	check(base) and $D_1 \circ \ldots \circ D_{m-1}$
2.	check(C_{m-1})	$D_1 \circ \ldots \circ D_{m-2}$ and check(C_m)
	$\vdots$	
m.	check(C_1)	$D_1 \circ \ldots \circ D_{base}$ and check(C_2)

Finally, use check(C_1) to discharge constraints on the base product. If the constraint on a property φ holds in the initial state of the base product, then φ holds of the composed system. If a validity check on φ fails to return true at the initial state of some feature, there may exist a path that fails to satisfy that property; potential feature interactions are reported in this instance.

Subtlety When computing constraints and data environments for the base, we divide the base into the portions that precede and follow the introduction of new features; removing the edges between the interface states of the base accomplishes this. No features follow the final states in the base product, so generating check(base) amounts to standard CTL model checking.

4.5. Where Does Verification Actually Happen?

When a system violates a property, it might do so in one of three different scenarios. First, a feature implementation is inherently incorrect, and the error can be detected by analyzing that feature alone. Second, the feature implementation is incorrect in the presence of some but not all collaborating modules. Third, each of the feature implementations is valid, but their composition interacts in a way that violates a system property. The second and third scenarios differ in whether the error is in a particular module (perhaps because it made assumptions that work in some situations but not in others) or only in their composition.

Constraint generation detects errors of the first kind. It traverses each feature to derive the assumptions under which the property holds; if the feature itself violates the property, constraint generation will return false. This is akin to property violation in traditional model checking.

Errors of the second and third kind do not get detected until constraint discharge. If discharging completes with all constraint checks returning true, then the property holds over the composed system; a result of false indicates an error. A result of $\top$ means there are paths on which the property does and does not hold, but this does not always indicate an error (e.g., for existentially-quantified path properties), and thus requires user interaction. Constraint discharge should not return $\perp$.

4.6. Soundness

The proposed methodology is sound if using it to verify a property yields the same result as verifying the property with standard model checking in the initial state of the composed system. The heart of the argument is that checking a constraint at a particular state of a feature F under a given data value V (Definition 4) yields the same result as verifying the constraint in that state in an augmented feature $F_V{}'$ that sets values of propositions according to the data value. Such a result defines how properties would be evaluated in the composed system, where all data propagations occur naturally and there is no need for a temporal constraint because the entire state space is available at analysis time. This argument (formalized in the following theorem) summarizes the overall soundness proof. The details, including proofs, are in the technical report.

Theorem 1 *Let F_1 and F_2 be features, s be a state in F_1, and φ a* CTL *formula. Let V be a data value coming into F_1. Let c be the result of* CONSTRAIN(F_1,φ,s). *Let c' be c with every annotated formula ψ_{s_t} replaced with the value of ψ (from a boolean lattice) in the initial state of F_2. Then $F_{1V}{}' \circ F_2, s \models \varphi$ if V satisfies c'.*

5. Experimental Study

We have implemented the methodology described in this paper and tested it on features and properties from Hall's email case study [22]. Our experiment was intended to determine whether the parameterized constraints were sufficient for modularly predicting the results of verifying properties in the composed product; in other words, we wanted to test how often the incompleteness of our approach affected verification in practice.

Because our algorithm is different from ordinary model checking, we cannot reuse an off-the-shelf model checker. We have therefore implemented our own prototype checker. Since the checker is a prototype built as a proof-of-concept, the performance numbers are not very meaningful. Nevertheless, the performance would be similar to that of a model checker, due to the deep structural similarity between a CTL model checker and our CONSTRAIN algorithm.

Hall's case study contains ten features such as mail delivery, digital signing and anonymous remailing. The requirements we verified are generated from his findings, and as such do encode some explicit checks for feature interaction:

1. Once a message is signed, the sender field is not altered until the message is delivered or received.

2. When a message is ready to be remailed, it is never mailed out with the sender's identity exposed.
3. If a receiver tries to verify a signature, then the message must be verifiable.
4. When a message is encrypted, it is never decrypted and then sent in the clear.
5. If a message is to be remailed, it is formatted correctly for the remailer to process it.
6. If an auto-response is generated, the response eventually is delivered or received.
7. If a message is forwarded, it is eventually delivered or received.
8. If the auto-responder replies to a message, then that message's subject line must be in the clear.
9. If an outgoing message is signed, its body is never changed unless is it delivered or retrieved.
10. If a mailhost generates an error message, then that message is eventually retrieved or delivered.

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. These properties are therefore useful for testing a modular technique.

Our experiment was successful, in that:

1. we correctly detected that the system failed each of these properties,
2. error detection required no traversals of the features beyond the constraint-generation phase, and
3. constraint discharge flagged the errors through propositional checks.

Some properties did yield $\top$, demonstrating the technique's incompleteness; in each case, inspection revealed an error.

6. Perspective and Future Work

Our case study shows that our new technique supports modular verification at least as well as our prior, three-valued, technique [28]. The following table shows the key differences:

	Prior	New
Traversals per interface generation	6	1
Persistent data propositions handled	weakly	yes
Properties lifted to entire system	no	yes
Each feature traversed per property	no	yes

The parameterized interfaces in the new technique reduce the number of state machine traversals required. The old technique avoided the subtleties in handling persistent data propositions by reducing open propositions to $\perp$; the new technique handles them directly with data environments.

The new technique appears inferior to the old technique only in the last line of the table, but this disadvantage is related to the advantage in the third line. The old technique did not always need to traverse each feature per property because the goal in the prior work was to detect when one feature violated properties true in the initial state of another: it did not attempt to "lift" the properties of one feature to the (initial state of the) entire composed system. In this work, we have shifted our attention to proving system-wide properties. Our algorithm determines whether a property holds in the initial state of the entire composed system. Traversing each feature per property is an unoptimized way to accomplish this lifting. In practice, we believe we can optimize this by making some of the traversals less expensive (e.g., checking reachability instead of computing constraints).

Traversing each feature per property appears to defeat the benefit of modular verification, which is usually to avoid traversing the entire state space. Our previous comment about optimization speaks to this issue, but there is a more fundamental answer. In a product-line context, modular verification avoids traversing each feature per property *per composed product*. Given that a set of features can yield an exponential number of products, limiting state space traversal to once per feature represents a significant cost savings over naive verification. Our algorithm provides this.

Our work does make some simplifying assumptions that we intend to address. First, while features may contain cycles internally, the graph of connections between features must form a DAG. This is less of a restriction than it seems, because feature compositions often take the form of pipe-and-filter systems. Indeed, our architecture is very similar to a version of the Jackson-Zave DFC model of features [23] restricted to static composition, and is therefore useful for modeling a wide variety of systems. (The systems used in case studies by Batory's group, such as FSATS [6], also obey this model.) Also, the CONSTRAIN algorithm assumes that no cycle within a feature sets the value of a data proposition (it handles all other internal cycles without imposing any restrictions—thus, for instance, it can freely handle systems with assignments to local data such as loop counters). We could relax this by setting the proposition's value to $\perp$, but our case study did not require it.

7. Related Work

There is a significant body of work on open system verification. The openness in prior work stems from both uncertainty in transitions and ignorance of propositions. Kupferman, Vardi and Wolper address the former [26]. Their work considers the failure of properties due to values generated by environment models. In particular, their methodology requires a property to hold in all environments; it does not classify the environments in which a property fails to hold.

In features, however, many property violations arise in only some contexts but not all. The Kupferman, et al. approach is therefore too restrictive in this setting.

Bruns and Godefroid consider properties that arise from partial Kripke structures, therefore having propositions of unknown value [9, 10]. They use a three-valued logic to preserve properties of the partial system in the complete structure. They handle the lack of parameterization under three-valued logic by performing two model checks on formulas with unknown values, one assuming all $\perp$ values are true (*optimistic*) and one assuming all $\perp$ values are false (*pessimistic*). A generalization of Bruns and Godefroid's technique is to use multi-valued model checking, pioneered by Chechik, Easterbrook and Devereaux [14]. Neither of these bodies of work discusses compositional verification.

Compositional verification is a well-explored idea [1]. Most of this work, however, examines the problem for parallel composition and assumes that composition does not add behavior to modules. The sequential composition model of features violates this assumption. Furthermore, few explicitly handle open systems in which the existence of propositions is unknown at module analysis time and almost none consider how to generate interfaces, as we do.

Giannakopoulou, Păsăreanu and Barringer [19] generate automata as interfaces for labeled transition systems under parallel composition. Neither of our approaches subsumes the other due to differences in composition model, properties handled and choice of system model (event- versus state-based). Houdini [18] infers annotations for ESC/Java, but these annotations are not property-driven and the approach is not truly modular.

Some work considers modular model checking under sequential control flow [2, 27]. Those efforts focus on making verification of a single system more tractable. Our work targets the plug-and-play world of product-lines, which requires constraint and interface generation rather than just model checking. None of those efforts explicitly handle the open systems and data persistence problems discussed here.

Our approach to constraint generation resembles temporal query checking, originally due to Chan [12]. Chan's approach assumed one variable per temporal logic formula and instantiated it with a propositional formula over variables in the model. Gurfinkel et al. [20] and Bruns and Godefroid [11] support multiple variables but still generate propositional constraints over model variables. Our work generates temporal constraints over propositions that are not in the model (since features are open systems). Our temporal constraints use subformulas of a given property formula; this restricted context enables temporal constraint generation in open systems.

Our methodology detects a special case of feature interaction errors [3, 8, 24], corresponding roughly to what Hall calls Type II interactions [21]. None of the other cited approaches detect interactions compositionally. Chechik and Easterbrook reason about compositions of concerns using multi-valued model checking [13]. Their framework identifies which concern (feature) is responsible for property violations when checking composed systems, but does not address proving properties through compositional reasoning.

Reussner [31] gives a theory of parameterized contracts. These contracts recognize that, when reusing a large component, different clients will need only parts of it; correspondingly, clients should need to satisfy only a part of the component's precondition. To support these scenarios, this technique specializes interfaces at composition time, employing automata-theoretic algorithms to compute these interfaces. In this respect, it is related to the constraints we derive. However, it assumes the existence of a fairly complete (manual) description of the component's behavior as the starting point for specialization, rather than exploiting the properties to automatically generate the interfaces.

8. Summary

This paper presents a compositional methodology for verifying features as open systems. By definition, any technique that attempts to verify open systems modularly must contend with insufficient information. The technique in this paper exploits a key insight about the nature of openness in compositional feature verification: openness arises from both propositional values flowing into a feature and temporal constraints on the control flow leaving a feature.

Concretely, this paper presents an algorithm that derives parameterized interface information to account for openness. We employ a flow analysis to derive a propositional formula summarizing the data values that each feature provides to subsequent features; using a variant of model checking, we derive a temporal constraint on the successor states of each feature. A series of simple propositional checks on the resulting constraints at composition time determines whether compositions of features violate system-wide properties. This approach is compositional because the latter checks rely only on the generated information, and do not re-visit the innards of individual features.

This technique improves on prior approaches that employ three-valued model checking to address openness. By separating the sources of openness, we are able to limit the use of three-valued reasoning to the propositional data and to composition time alone. This leads to interfaces that are simpler and more accurate without a large explosion in their size. In addition, our technique performs only propositional calculation, not model checking, at composition time, making this a lightweight step.

Acknowledgment We thank the anonymous reviewers for their extremely careful reading of the paper, which clarified several issues and improved the presentation.

References

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

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

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

[4] Aspect oriented programming (article series). *Communications of the ACM*, 44(10), Oct. 2001.

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

[6] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific la nguages: A case study. *ACM Transactions on Software Engineering and Methodology*, April 2002.

[7] C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. A constraint-based approach to open feature verification. Technical Report CS-03-07, Brown University Department of Computer Science, 2003.

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

[9] 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.

[10] 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.

[11] G. Bruns and P. Godefroid. Temporal logic query checking. In *IEEE Symposium on Logic in Computer Science*, pages 409–417. IEEE Press, 2001.

[12] W. Chan. Temporal-logic queries. In *International Conference on Computer-Aided Verification*, pages 450–463, 2000.

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

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

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

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

[17] 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.

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

[19] D. Giannakopoulou, C. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *IEEE International Conference on Automated Software Engineering*, pages 3–12, 2002.

[20] A. Gurfinkel, B. Devereux, and M. Chechik. Model exploration with temporal logic query checking. In *Symposium on the Foundations of Software Engineering*. ACM Press, 2002.

[21] R. J. Hall. Feature combination and interaction detection via foreground/background models. In *Feature Interactions in Telecommunications Systems*. IOS Press, 1998.

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

[23] 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.

[24] 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.

[25] G. A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206, 1973.

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

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

[28] H. C. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features through three-valued model checking. *Automated Software Engineering: An International Journal*, 2003.

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

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

[31] R. H. Reussner and H. W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In *ICSE Workshop on Component-Based Software Engineering*, Apr. 2002.

[32] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[33] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, Dec. 1999.