A Constraint-Based Approach to Open Feature Verification

Colin Blundell, Kathi Fisler, Shriram Krishnamurthi, and Pascal van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-03-07
May 2003

A Constraint-Based Approach to Open Feature Verification

Colin Blundell, Kathi Fisler, Shriram Krishnamurthi, and Pascal van Hentenryck

May 19, 2003

Abstract

Feature-oriented software architectures provide a powerful model for building product-line systems. Each component corresponds to an individual feature, and a composition of features yields a product. Feature-oriented verification methodologies must be able to analyze individual features and to compose the results into results on products; features are hence a form of open systems. In prior work, Li, Fisler and Krishnamurthi proposed a feature verification methodology based on 3-valued model checking. This paper presents a new methodology based on constraint-generation that is simpler and less expensive than the original. In addition, it supports both client-side and producer-side notions of compositional feature verification.

1 Introduction

Feature-oriented architectures organize code around the features that a system contains, rather than the objects that form the core of its implementation. 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 [10], and provides a meaningful framework for component [15, 22] reuse.

While there is growing support for development around features [2, 3, 11, 19, 20, 21, 23], most of this work ignores key questions of static analyses leading to forms of verification. In principle, feature-orientation should simplify verification because both features and requirements arise from a user's view of a system. Ideally, we would like to verify requirements against the individual feature modules that implement them, then use some composition-time checks to confirm that features do not interfere with each other's properties in composed systems. In practice, feature-oriented verification is challenging because features interact through issues such as shared state or inconsistent requirements. (This is an important special case of the feature interaction problem [4, 16].)

Li, Fisler and Krishnamurthi have previously argued that any modular view of feature verification must treat features as *open systems* to support data that propagate across features [18]. This requires techniques for handling propositions whose values are not available at analysis time. Their work used three-valued model checking as a foundation for compositional verification of open features. While effective at detecting property violations compositionally, this approach had several drawbacks: it required multiple model checking passes and multiple interfaces to handle different combinations of three-valued variables, and generally seemed more cumbersome than necessary.

An interesting observation emerges from a closer look at openness in the context of feature verification. Features are open with respect to two different kinds of information: propositional values flowing into a feature (representing data attributes) and the successor states and paths that follow the feature. Treating these two forms of openness separately leads to a new methodology for compositional feature verification. This paper presents such a methodology and demonstrates that it is simpler and more lightweight than the methodology in [18], while matching its performance on a case study. Our approach uses a combination of data flow analysis and constraint generation. Handling the two forms of openness separately allows our new methodology to rely on two-valued rather than three-valued model checking; as a result, the algorithm performs fewer model checking passes and generates more natural interfaces. In addition, this result has a novel feature not found in previous work: it can be applied in two different contexts, corresponding roughly to the distinction between consumers and producers of components, thereby offering two different cost models for its use.

The rest of this paper is organized as follows. Section 2 gives an overview of the feature verification problem and our constraint-based approach. Section 3 presents related work. Sections 4 and 5 present our formal model of features and the details of our methodology. Section 6 discusses our experimental validation. Section 7 presents our analysis of the proposed approach and outlines future work. Section 8 offers concluding remarks.

2 Overview of Approach

Consider an email feature suite, which includes components for anonymous remailing and digital signing of messages.[1] A product might include these two features and basic mail delivery, as shown in Figure 1.

This product should satisfy a requirement that if a message is marked for anonymous remailing, then it should not be digitally signed before it is mailed. The temporal logic formula $\mathsf{AG}(\mathit{remailed} \rightarrow \mathsf{A}(\neg \mathit{signed}\ \mathsf{R}\ \neg \mathit{mailed}))$ captures this property. This property checks that the email product supports the goal of remailing: to maintain the anonymity of the sender (which digital signing would circumvent). Failure of this property would indicate an instance of feature interaction between remailing and

[1] These examples come from a suite due to Robert Hall [14].

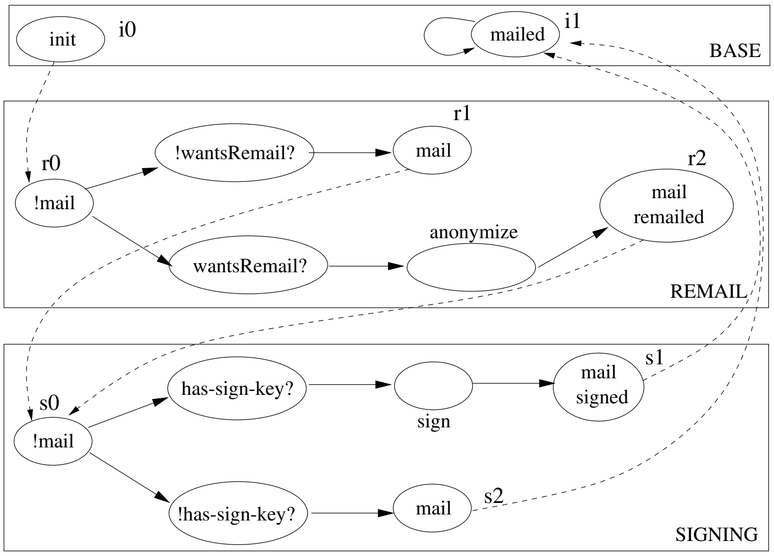

Figure 1: A simple email product with remailing and message signing. The dashed transitions show one possible assembly of features into a composed product. Another assembly might permute the order of features. In the state machines, ! denotes logical negation, propositions ending in ? represent control decisions, and all other propositions represent data attributes of email messages. Identifiers next to states simply name those states.

digital signing. The sample product in Figure 1 can violate this property on a path that includes the upper path of states in the SIGNING feature.

A compositional verification methodology would be able to evaluate this property against the product using a series of independent checks on the individual features. If the methodology requires traversing the composed product (potentially expensive or even intractable if the product is large), then the methodology is non-compositional. This paper presents a compositional methodology for open feature verification using the CTL model checking algorithm [9] as its basis.[2]

The standard CTL model checking algorithm does not inherently support the kinds of features we analyze. For instance, if a CTL model checker evaluated the given property in the initial state of the REMAIL feature alone, it would report the property as true because REMAIL does not mention the proposition *signed* (the model checker would therefore assume that *signed* is false). This is erroneous in two ways. First, with features being open components, a model checker cannot assume that propositions are false simply because they do not appear in a feature or in a particular state: the algorithm must assume that some propositions may be asserted prior to executing a feature and that those values should persist until explicitly unset. Second, with features containing only a portion of the entire product's state space, the model checker lacks information about the properties that hold along paths that emanate from the feature. It is possible to use REMAIL in two different products such that one satisfies this property and one does not. At best, a model checker can only traverse the feature and determine what constraints it requires on the features that eventually connect to it.

In this particular example, in order for the property to hold in the composed system, any successor to state r_2 in REMAIL must eventually reach the *mailed* state (to discharge the release operator) unless the *signed* attribute is set to false. The temporal logic formula $\mathsf{A}(\neg signed \; \mathsf{R} \; \neg mailed)$ captures this constraint. Whether the original property holds in the SIGNING feature will depend on the value of *remailed* that propagates from REMAIL to SIGNING at composition time. To remain compositional, the methodology must compute information about the values of propositions that one feature can propagate to subsequent features. In this example, the REMAIL feature has a path on which the proposition *remailed* is true.

Intuitively, the overall methodology will generate data bindings from and constraints on each feature, then use the bindings and constraints to discharge constraints on other features; in other words, the constraints on a feature will be parametric in information from the features that both precede and follow that feature. The methodology will attempt to discharge each constraint assuming that (a) all data values from earlier features have propagated to the current feature, and (b) that the temporal constraints have been discharged on all subsequent features. Certain failed attempts to discharge constraints will indicate failures of properties in the composed system; properties hold in the composed system if all of their constraints discharge successfully. The rest of this paper presents this methodology more formally.

[2]This paper assumes familiarity with CTL.

3 Related Work

Our approach to constraint generation resembles temporal query checking, originally due to Chan [6]. 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. [13] and Bruns and Godefroid [5] 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 are subformulas of a given property formula; this restricted context enables temporal constraint generation in open systems. If we were interested in generating propositional constraints on open systems, we could use Chechik and Easterbrook's multiple valued logic framework [7], but the resulting algorithm would be doubly exponential in the number of unknown propositions and would not naturally extend to temporal constraints.

Our methodology provides a restricted form of feature interaction detection [1, 4, 16], in which interactions arise as violations of temporal logic properties. None of the other cited approaches detect interactions compositionally. Chechik and Easterbrook reason about compositions of concerns using multi-valued model checking [8]. Their framework identifies which concern (feature) is responsible for property violations when checking composed systems, but does not address proving properties through compositional reasoning.

4 A Formal Model of Features and Their Compositions

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. Fisler and Krishnamurthi have previously showed how to reduce models where each feature has multiple state machines to the single-machine model [12], hence we adopt the single-machine model here for simplicity.

In this section, the notation $PL(\phi)$ denotes the set of propositional logic expressions over the set of propositions in ϕ.

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,
- $Tr : S \rightarrow 2^{\Delta}$ indicates which propositions are set to true in each state, and $Fa : S \rightarrow 2^{\Delta}$ indicates which propositions are set to false in each state ($\forall s \in S, Tr(s) \cap Fa(s) = \emptyset$).

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 necessarily imply its falsity. Our model therefore employs distinct labeling functions for true and false labels.

Our methodology distinguishs between two different uses of propositions within a state machine: control propositions and data propositions [17]. 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*. Our methodology treats data propositions as persistent: their values hold in a system until explicitly changed by an assignment in another state. Control propositions are not persistent: the labelling functions indicate values of control propositions on a per-state basis (and are assumed false) if no explicit labelling exists. Our model requires the designer to classify propositions as data or control within a feature (as shown in definition 4).

We expect features within a system to compose in a pipe-and-filter fashion 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). A composition of features and the base infrastructure forms a *product*, where a product consists of both a state machine and a set of interfaces where new features may be inserted into the system. We capture the common infrastructure in a *base product*, which is like a core feature.

In order to view base products and features as components for compositional verification, each requires a notion of an interface. In this model, the interfaces simply specify those states to which other features connect via both outgoing and incoming edges. In the REMAIL feature from Figure 1, for example, the interface would specify that edges leave from states r_1 and r_2, (the *outgoing* interface) and enter at r_0 (the *connect* interface).

Intuitively, composing features and products involves adding edges between interface states. Adding all possible such edges, however, might merge paths that should otherwise remain distinct. A more complex email product, for example, might have different paths through a feature for messages to be delivered on the local network versus to another network. Our interfaces therefore include connection specifications for outgoing edges from a state machine. These specifications constrain target states to satisfying certain propositions. The following definition formalizes connection specifications and when they are compatible with potential target states:

Definition 2 Given a state machine, a *connection specification* is a set of tuples $\langle s_e, g, t, f \rangle$ where $s_e \in S$, $g \in PL(\Sigma)$, and t and f are subsets of 2^{Δ}. Connection specifications must require edges added from s_e on the basis of the specifications to be deterministic. A state s (potentially from another state machine) is *compatible* with $\langle s_e, g, t, f \rangle$ if every $p \in t$ is in $Tr(s)$ and every $p \in f$ is in $Fa(s)$. Compatibility indicates that composition may insert an edge from s_e to state s; this edge would have guard g from the specification.

We now define base products, features, and their interfaces.

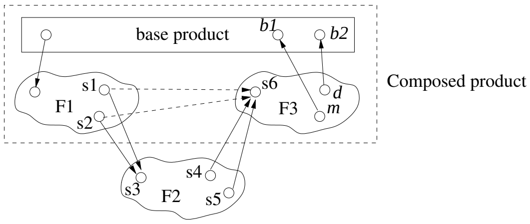

Figure 2: Inserting a feature into a product.

Definition 3 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} (this represents the system with no features).

Definition 4 A *feature* hereafter will be a state machine

$$(S, \Sigma, \Delta, S_0, R, Tr, Fa)$$

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 model supports two techniques for combining features: features can be composed into compound features, and features (composed or simple) can be inserted into products (to form new products). Our methodology uses both techniques, as defined below.

Definition 5 Let F_1 and F_2 be features with respective interfaces $\langle S_{\text{in}_1}, S_{\text{exit}_1}, R_{\text{exit}_1}\rangle$ and $\langle S_{\text{in}_2}, S_{\text{exit}_2}, R_{\text{exit}_2}\rangle$. The composition of F_1 followed by F_2 (denoted $F_1 \circ F_2$) is a feature with state machine $(S, \Sigma, \Delta, S_0, R, Tr, Fa)$, data propositions D, and interface $\langle S_{\text{in}_1}, S_{\text{exit}_2}, R_{\text{exit}_2}\rangle$ where

- S, Σ, Δ, Tr, Fa, and D are the unions of their respective counterparts in F_1 and F_2.
- R is the union of the transition relations in F_1 and F_2, plus all edges (s_e, g, s_i) such that $s_e \in S_{\text{exit}_1}$ and s_i is compatible with some $\langle s_e, g, t, f\rangle$ in R_{exit_1}.

Inserting a feature into a product is more complicated, as it also involves removing certain edges so that control routes through the new feature. The removed edges, shown as dashed lines in Figure 2, connect interface states of the product; they are replaced by paths through the new feature.

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

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

Let F be a feature with state machine

$$(S_F, \Sigma_F, \Delta_F, S_{0_F}, R_F, Tr_F, Fa_F)$$

and interface $\langle S_{\text{in}}, S_{\text{exit}}, R_{\text{exit}}\rangle$. The composition of P and F via interface $\langle S_{\text{outgoing}_i}, S_{\text{connect}_i}\rangle$ is a product P_C. The state machine component of P_C is

$$(S_C, \Sigma_C, \Delta_C, S_0, R_C, Tr_C, Fa_C)$$

where

- S_C, Σ_C, Δ_C, Tr_C, and Fa_C are the unions of their counterparts in P and F,
- $R_C = R_P \cup R_F$ except all edges between the interface states S_{outgoing_i} and S_{connect_i} from R_P and with
  - Every edge (s_o, g, s_i) such that $s_o \in S_{\text{outgoing}_i}$, $s_i \in S_{\text{in}}$, and s_i is compatible with some tuple $\langle s_o, g, t, f\rangle \in R_{\text{out}_i}$.
  - Every edge (s_e, g, s_c) such that $s_e \in S_{\text{exit}}$, $s_c \in S_{\text{connect}_i}$, and s_c is compatible with some tuple $\langle s_e, g, t, f\rangle \in R_{\text{exit}}$.

The interfaces of P_C is the set of interfaces from P except interface $\langle S_{\text{outgoing}_i}, S_{\text{connect}_i}, R_{\text{out}_i}\rangle$ and augmented with two new interfaces $\langle S_{\text{outgoing}_i}, S_{\text{in}}, R_{\text{out}_i}\rangle$ and $\langle S_{\text{exit}}, S_{\text{connect}_i}, R_{\text{exit}}\rangle$.

Figure 2 illustrates these definitions more intuitively. The figure shows a product consisting of a base product and two features F_1 and F_3, and the 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 and adds the four edges that connect F_2 to the product. The italic labels m and d on the states of F_3 and the base product capture the idea behind states being compatible on composition. When F_3 was composed with the base product, edges connected the states from F_3 to the base based on constraints in R_{exit} (not shown).

5 The Compositional Verification Methodology

Section 2 gave an intuitive description of our methodology; this section presents the formal details. The algorithm consists of two phases: an initial analysis phase that generates constraints and produces data environments for each feature, and an assembly phase that confirms that each constraint on a feature holds given the data environments of all preceding features and assuming that all constraints on subsequent features have been discharged.

Section 5.1 presents the algorithm for generating the temporal constraints for a feature. Section 5.2 defines data environments. Section 5.3 describes the algorithm for discharging constraints at assembly time. The presentation continues to use the sample product from Figure 1 as a running example.

5.1 Generating Constraints

Given a feature and a property (as a CTL formula), the constraint generation algorithm generates a constraint that is parameterized over both the incoming data environment and the paths emanating from the terminal states of the feature. The constraint takes the form of a specialized form of CTL formula: the formula uses names of propositions to parameterize over the data environment and tagged subformulas to parameterize over the exit paths. The tags on the subformulas contain names of terminal states in the feature; a tag denotes that the subformula must hold in the successor of the state named in the tag for the property to hold in the composed system (this successor is determined only at composition time). In our examples, we denote the tags by subscripts on the subformulas.

As 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. Running the constraint generation algorithm on φ at the initial state (r_0) of REMAIL yields the annotated formula

$(\varphi_{r_1} \wedge (\neg \mathit{remailed} \vee (\mathit{remailed} \wedge (\mathsf{EF}\mathit{mailed})_{r_1}))) \wedge (\varphi_{r_2} \wedge (\mathsf{EF}\mathit{mailed})_{r_2})$

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 eventually reaches the *mailed* state from r_1 unless *remailed* is already false. This arises from the implication being tested at r_0. The constraint is simpler for r_2, because a path to r_2 is known to satisfy the *remailed* proposition.

Figure 3 shows the constraint-generation algorithm (called CONSTRAIN). The algorithm mostly follows the semantic definition of CTL: given the property to constrain at the initial state, it decomposes the property into constraints on subformulas in successor states. Intuitively, the algorithm partially evaluates the given property over the feature, under the assumption that data propositions persist along paths. The *path-env* argument to CONSTRAIN handles persistence by storing the most recent value of each data proposition along the current path. The *marked* argument to CONSTRAIN contains states that have already been visited (used to terminate the recursion).

CONSTRAIN diverges from the normal CTL semantics as necessary to parameterize over data environments and terminal state properties during this partial evaluation. For example, in the propositional case, the result of the CONSTRAIN algorithm depends on the nature of the proposition relative to the feature.

- 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 are not persistent, and hence do not appear in *path-env*.
- If the proposition is a control proposition of another feature (indicated by the proposition being in the set *C-external*), its value must be false in this feature. This situation can arise when checking a property that is primarily about one feature in another feature; the

The EXTEND-PATHENV function referenced in these algorithms takes a path-environment and a state s. For all data propositions p assigned values in $Tr(s)$ or $Fa(s)$, EXTEND-PATHENV updates the path-environment to reflect the corresponding value of p at s.

The CONSTRAIN Algorithm

CONSTRAIN takes a CTL-formula, a state, and a mapping from propositions to boolean (a persistent valuation called the *path environment*). The algorithm returns a constraint formula. In the algorithm, p denotes an atomic proposition, s denotes a (non-terminal or terminal) state, s_n denotes a non-terminal state, s_t denotes a terminal state, *C-external* is a set of control propositions that are external to the feature, and ϕ, ϕ_1, ϕ_2 are CTL formulas.

CONSTRAIN($p, s,$ *path-env*) = true if p is a data proposition of the current feature and *path-env*(p) = true
false if p is a data proposition of the current feature and *path-env*(p) = false
false if $p \in$ *C-external*
true if p is not a data proposition and $p \in Tr(s)$
false if p is not a data proposition and $p \in Fa(s)$
p otherwise (i.e., p is a data proposition from another feature)

CONSTRAIN($\neg\phi, s,$ *path-env*) = $\neg$CONSTRAIN($\phi, s,$ *path-env*)

CONSTRAIN($\phi_1 \lor \phi_2, s,$ *path-env*) = CONSTRAIN($\phi_1, s,$ *path-env*) $\lor$ CONSTRAIN($\phi_2, s,$ *path-env*)

CONSTRAIN($\phi_1 \land \phi_2, s,$ *path-env*) = CONSTRAIN($\phi_1, s,$ *path-env*) $\land$ CONSTRAIN($\phi_2, s,$ *path-env*)

CONSTRAIN($\mathsf{EX}\phi, s_n,$ *path-env*) = $\bigvee_{t \in post(s_n)}$ CONSTRAIN($\phi, t,$extend-path-env(*path-env*$, t$))

CONSTRAIN($\mathsf{EX}\phi, s_t$) = ϕ_{s_t}

CONSTRAIN($\mathsf{E}[\phi_1 \mathsf{U} \phi_2], s,$ *path-env*) = PATH-CONSTRAIN($\mathsf{E}[\phi_1 \mathsf{U} \phi_2], s, \emptyset,$ *path-env*)

CONSTRAIN($\mathsf{EG}\ \phi, s,$ *path-env*) = PATH-CONSTRAIN($\mathsf{EG}\ \phi, s, \emptyset,$ *path-env*)

The PATH-CONSTRAIN Algorithm

PATH-CONSTRAIN takes a CTL-formula, a state and a set of states already visited during constraint generation and a path environment and returns a CTL constraint formula. In the algorithm definition, s denotes a (non-terminal or terminal) state, s_n denotes a non-terminal state, s_t denotes a terminal state, *marked* is a set of states, and ϕ, ϕ_1, ϕ_2 are CTL formulas.

PATH-CONSTRAIN($\mathsf{E}[\phi_1 \mathsf{U} \phi_2], s_n, marked,$ *path-env*) =
False if $s_n \in marked$
otherwise: CONSTRAIN($\phi_2, s_n,$ *path-env*)$\lor$(CONSTRAIN($\phi_1, s_n,$ *path-env*)$\land$
$\bigvee_{t \in post(s_n)}$ PATH-CONSTRAIN($\mathsf{E}[\phi_1 \mathsf{U} \phi_2], t, marked \cup \{s_n\}$, extend-path-env(*path-env*$, t$)))

PATH-CONSTRAIN($\mathsf{E}[\phi_1 \mathsf{U} \phi_2], s_t, marked,$ *path-env*) =
CONSTRAIN($\phi_2, s_t,$ *path-env*) $\lor$ (CONSTRAIN($\phi_1, s_t,$ *path-env*) $\land$ ($\mathsf{E}[\phi_1 \mathsf{U} \phi_2])_{s_t}$)

PATH-CONSTRAIN($\mathsf{EG}\ \phi, s_n, marked,$ *path-env*) =
true if $s_n \in marked$
otherwise: CONSTRAIN($\phi, s_n,$ *path-env*)$\land$
$\bigvee_{t \in post(s_n)}$ PATH-CONSTRAIN($\mathsf{EG}\ \phi, t, marked \cup \{s_n\}$, extend-path-env(*path-env*$, t$))

PATH-CONSTRAIN($\mathsf{EG}\ \phi, s_t, marked,$ *path-env*) = CONSTRAIN($\phi, s_t,$ *path-env*) $\land$ ($\mathsf{EG}\ \phi)_{s_t}$

work of Li, Fisler and Krishnamurthi provides several such examples [17].

- If the proposition is neither a control proposition nor a data proposition of the feature being verified, nor is in *C-external*, then its value must come from the incoming data environment. The CONSTRAIN algorithm inserts the proposition itself into the constraint formula (as seen in the last line of the propositional case) to parameterize the constraint over the value from the data environment.

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; in Figure 3, this happens in the cases of CONSTRAIN and PATH-CONSTRAIN that process terminal states (denoted s_t).[3]

The PATH-CONSTRAIN algorithm in Figure 3 is an auxiliary function in CONSTRAIN; it is called to process properties that traverse paths (i.e., those involving the EU and EG operators).

5.2 Data Environments

Previous sections motivated the persistent nature of data propositions. Since data persists across states and across features, each feature eventually operates in the context of values for propositions that are set by previous features. We refer to this information as a *data environment*. Each feature produces a data environment for subsequent features that summarizes the values that the feature assigns to propositions. This section formally defines data environments. We determine data environments using a computation that closely resembles constant-propagation through flow analysis.

Definition 7 *Let F be a feature with data propositions D. 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. *A proposition p is* defined *for Π if $p \in D$ and there exists a state s_i such that $p \in Tr(s_i)$ or $p \in Fa(s_i)$. Def(Π) denotes the set of all p that are defined for Π.*
2. *Let $p \in$ Def(Π). The* value *for p in Π, denoted val(p, Π), is* true *(resp.* false*) if there exists a state s_i such that p in Tr(s_i) (resp. Fa(s_i)) and p is not in Def(s_j) for some $j > i$.*
3. *The* data value *for Π is the set $\{\langle p, val(p, \Pi)\rangle | p \in Def(\Pi)\}$.*
4. *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.*[4] *We overload the term data environment to refer also to a function from terminal states to their individual data environments.*

[3]This tagging is unnecessary in the propositional case because incoming data propositions hold their values unless explicitly set.

[4]We assume a standard fixpoint construction to handle infinitely many paths.

For the REMAIL feature shown in Figure 1, the data environment would map state r_1 to $\langle r_0, \{\langle mail, \mathsf{true}\rangle\}\rangle$ and would map state r_2 to $\langle r_0, \{\langle mail, \mathsf{true}\rangle, \langle remailed, \mathsf{true}\rangle\}\rangle$. The data environment ignores the setting of *mail* to false in state r_0 because both paths override that definition later in the feature. The proposition *wantsRemail?* does not appear in the data environment because it is a control proposition.

When computing the data environment for the base product, the methodology must first remove 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.

5.3 Verifying Products

Sections 5.1 and 5.2 define the fundamental building blocks for compositional feature verification based on constraints: annotated CTL constraint formulas and data environments. We present two different approaches to compositional feature verification that build on these techniques. This section presents the assembly-time algorithm that we have described intuitively earlier in the paper; Section 5.4 outlines the other.

The assembly-time methodology proceeds in two stages: first, it composes the data environments coming into each feature with the data environments from the preceding features; second, it checks the constraints on terminal states using the previously checked constraints from subsequent features. Assume $F_1, \ldots, F_n$ denote features, D_i the data environment that F_i produces, C_i the constraint on F_i and check(C_i) the result of checking constraint C_i. Informally, the following equations summarize the methodology:

- Compute check(C_n) using check(base) and $D_1 \circ \ldots \circ D_{n-1}$
- Compute check(C_{n-1}) using $D_1 \circ \ldots \circ D_{n-2}$ and check(C_n)
- ...
- Compute check(C_i) using $D_1 \circ \ldots \circ D_{i-1}$ and check(C_{i+1})
- Check initial base constraint using check(C_1)

Section 5.3.1 describes how to compose data environments and Section 5.3.2 presents the overall algorithm.

5.3.1 Composing Data Environments

When two features F_1 and F_2 are composed sequentially within a product, the data environment produced by F_1 augments the data environment produced by F_2; this reflects that F_2 may be open with respect to propositions from F_1. A compositional verification methodology must be able to compose data environments without traversing the composition of their features. This section presents data environment composition at a definitional level; in our actual implementation, we use a particular data structure for data environments that enables their simple and efficient composition (we defer the implementation details to a longer paper).

Definition 8 *Let F_1 and F_2 be features with data environments E_1 and E_2, respectively. Let R be the set of all transitions from states from F_1 to states of F_2 in the composition of F_1 and F_2. Let s_{t_2} be a state in the domain of E_2 (i.e., a terminal state of F_2). The* composed data environment *for s_{t_2} under R is the set of all pairs $\langle s_{0_1}, \{V_1, \ldots, V_k\}\rangle$ such that there exists*

- *$\langle s_{0_1}, \{DV1_1, \ldots, DV1_n\}\rangle$ in $E_1(s_{t_1})$,*
- *a transition (s_{t_1}, s_{0_2}) in R, and*
- *$\langle s_{0_2}, \{DV2_1, \ldots, DV2_m\}\rangle$ in $E_2(s_{t_2})$*

and for every h in $1 \ldots k$, there exists $DV1_i$ and $DV2_j$ such that V_h is $DV2_j \cup \{\langle p, v\rangle \in DV1_i | p \notin dom(DV2_j\}$ (in other words, all bindings from $DV1_i$ whose propositions are not bound in $DV2_j$). The data environment composed from E_1 followed by E_2 via R, *denoted $E_1 \circ_R E_2$, is a function from states s_t in the domain of E_2 to the composed data environment for s_t under R.*

In our running example (Figure 1), the REMAIL feature's data environment maps state r_1 to $\langle r_0, \{\langle mail, \mathsf{true}\rangle\}\rangle$ and state r_2 to $\langle r_0, \{\langle mail, \mathsf{true}\rangle, \langle remailed, \mathsf{true}\rangle\}\rangle$. The SIGNING feature's data environment maps s_1 to $\langle s_0, \{\langle mail, \mathsf{true}\rangle, \langle signed, \mathsf{true}\rangle\}\rangle$ and state s_2 to $\langle s_0, \{\langle mail, \mathsf{true}\rangle\}\rangle$. Composing these data environments via the dashed transitions in Figure 1 would map s_1 to

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

and state s_2 to $\langle s_0, \{\langle mail, \mathsf{true}\rangle, \langle remailed, \mathsf{true}\rangle\}\rangle$. This composition ignores the value of *mail* in REMAIL because *mail* is also set in SIGNING. If the SIGNING feature did not set the *mail*, the composed data environment would still set *mail* to true based on the bindings in the REMAIL feature.

The following two lemmas establish that composed data environments are equivalent to those derived from composed features.

Lemma 1 *Let F_1 and F_2 be features with data environments E_1 and E_2, respectively. Let R be the set of all transitions from states of F_1 to states of F_2 in the composition $F_1 \circ F_2$. Let $E'_{s_{t_2}}$ be the composed data environment for some s_{t_2} under R and $E_{s_{t_2}}$ be the data environment for s_{t_2} in the composition. $\langle s_{0_1}, DV\rangle \in E_{s_{t_2}}$ iff $\langle s_{0_1}, DV\rangle \in E'_{s_{t_2}}$.*

Proof: Appendix A.

Lemma 2 *Let F_1 and F_2 be features with data environments E_1 and E_2, respectively. Let R be the set of all transitions from states of F_1 to states of F_2 in the composition $F_1 \circ F_2$. Let E be the data environment of the composition $F_1 \circ F_2$. E is equivalent to the composed data environment $E_1 \circ_R E_2$.*

Proof:

For each s_{t_2} in the domain of the data environents, the data environment of s_{t_2} in the composition is equivalent to its composed data environment by Lemma 1.

The following result shows that Lemma 2 can be extended to systems with arbitrary numbers of features.

Corollary 2.1 *Let F_1, F_2, ... F_n be features with data environments E_1, E_2, ... E_n respectively. Let R_i be the set of all transitions from states of F_i to states of F_{i+1} in the composition $F_i \circ F_{i+1}$. Let E be the data environment of the composition $F_1 \circ F_2 \circ \ldots \circ F_n$ and E' be the composed data environment $E_1 \circ_{R_1} E_2 \circ \ldots E_{n-1} \circ_{R_{n-1}} E_n$. Then E is equivalent to E'.*

Proof:

We will prove this by induction.

Base case: $n = 2$.

In this case Lemma 2 gives the desired result.

Inductive step: Assume the corollary holds for $n-1$ features. Show that it holds for n features.

By the inductive hypopaper, we have that the data environment of the composition $S' = F_1 \circ F_2 \circ \ldots F_{n-1}$ is equivalent to the composed data environment $E' = E_1 \circ_{R_1} E_2 \circ_{R_2} \ldots E_{n-1}$. Therefore, we have by Lemma 2 that the composed data environment $E' \circ_{R_{n-1}} E_n$ is equivalent to the data environment of the composition $S' \circ F_n$, which gives the desired result.

5.3.2 The Verification Methodology

The methodology consists of two distinct phases: an initial analysis phase that derives data environments and constraints on all features and a composition phase that checks for constraint violation between features.

Initial Analysis Assume that a product family will be built from a set of m features and a base product for the family. Assume that the requirements for features in the family are known, and have been expressed as a series of CTL formulas $\varphi_1, \ldots, \varphi_k$. Prior to product composition time, assume that the product assembler has generated the data environments produced by each feature plus the base product, and run the CONSTRAIN algorithm on each feature for every subformula of each φ_i. These steps can be performed once for the set of candidate features for a product and reused over multiple assemblies from the same set of features and candidate requirements.

Composition Time Assume the client has chosen a sequence of n of the original m features to assemble into a product, composed the features in order (definition 5), then inserted them into the base feature (definition 6). The algorithm (VERIFY, shown in Figure 4) proceeds in two passes over the composed assembly:

1. (VERIFY, step 1) Working forwards from the base product through all of the features, compose the data environment produced by each feature with the data environment produced by the preceding feature.
2. (VERIFY, step 2) Working backwards from the base product through all of the features, instantiate the constraints on each feature with the values from the data environment and the results of each annotated subformula in the constraint from the subsequent feature.

The Verify Algorithm

Let P be a product composed from a base product B and features $F_1, \ldots, F_n$. Let φ be a CTL formula for a property that should hold of P. Let *ENVS* be a function from terminal states of features in P to their data environments (definition 7); for the outgoing interface states in B, *ENVS* should set all data propositions from $F_1, \ldots, F_n$ to false. The notation $ENVS|F_{i-1}$ denotes *ENVS* restricted to terminal states from feature F_{i-1}. $ENVS|S$ denotes *ENVS* restricted to terminal states in the set S.

Let *Constraints* be a set of tuples $\langle s_0, \psi, \text{CONSTRAIN}(s_0, \psi, bindings(s))\rangle$, where s_0 is an initial state of some feature or the base product in P, ψ is a subformula of φ, and $bindings(s_0)$ is the set of bindings to propositions in state s_0 from Tr or Fa. For each run of CONSTRAIN in the base product, remove all edges between the interface states prior to generating the constraint (for the same reasons as described when computing the data environment for the base).

Let *SUBS* be a function from initial states and subformulas of φ to the set $\{\mathsf{true}, \mathsf{false}, \perp\}$. This stores the results of checking each constraint under the composed data environments and the verified subsequent features.

1. For each feature F_i ($i = 0$ to n, where F_0 is the base feature) and each terminal state s_t of F_i, replace $ENVS(s_t)$ with the composition of $ENVS|F_{i-1}$ and $ENVS(s_t)$) via the set of transitions between features in P.
2. For each feature F_i ($i = n+1$ to 1, where F_{n+1} and F_0 refer to the base feature), each initial state s_0 of F_i, and each (s_0, ψ, c) in the domain of *Constraints*, check $\text{PRESERVED}(s_0, \psi, ENVS|pre(s_0))$.
3. If $SUBS(s_0, \varphi) = \mathsf{true}$, where s_0 is the initial state of P, then φ is true in P. If $SUBS(s_0, \varphi) \neq \mathsf{true}$ for initial state s_0 of some feature in P, then φ does not hold in P (this detects potential feature interactions).

The preserved Algorithm

PRESERVED takes an initial state s_0 of some feature, a formula φ and a data environment E and determines whether E satisfies the constraint for φ at s_0. Given a state s, $post(s)$ refers to the successors of s in the composed system. This method works as follows:

1. For each subformula ψ in $Constraints(s_0, \varphi)$ that is annotated with a state tag s_t, substitute $SUBS(post(s_t), \psi)$ for ψ in φ (this substitution is justified because annotated formulas reflect constraints on the successor states of those named in the tag, as discussed in Section 5.1). The resulting formula φ is completely propositional (but may include $\perp$).
2. For every data value V in E, substitute each proposition p in the resulting formula with its value in V and check the resulting formula for validity. If all such checks are valid, set $SUBS(s_0, \varphi)$ to true. If all such checks are invalid, set $SUBS(s_0, \varphi)$ to false. Otherwise, set $SUBS(s_0, \varphi)$ to $\perp$.

Figure 4: The assembly-time algorithm for discharging constraints.

Check the instantiated formula for validity (PRESERVED algorithm, Figure 4). Note that this step requires only substitution of previously computed results, not model checking; hence the approach is compositional. This step may, however, require three-valued propositional reasoning as subformulas may not be true in all data values in a data environment (hence the method is incomplete).[5]

3. (VERIFY, step 3) If the constraint on a property φ_i holds in the initial state of the base product, then φ_i holds of the composed system. If a PRESERVED check on one of the $\varphi_1, \ldots, \varphi_k$ 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.

Note that all model checking occurred in the CONSTRAIN checks of the initial analysis. The computations required at composition time are 3-valued propositional validity checks.

Soundness

The proposed methodology is sound if verifying a property in the composed system yields the same result as verifying the property using the algorithm presented in Section 5.3.2. This section presents the main definition and theorems needed to prove soundness.

The heart of the argument is that checking a constraint at a particular state under a given data value yields the same result as verifying the constraint in that state in an augmented feature that sets values of propositions according to the data value. Such a result defines the context in which 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, combined with the correctness lemma on composing data environments from Section 5.3.1, yield the overall soundness theorem.

The following definition describes how to augment a feature with the values given in a specific data value:

Definition 9 *Let F be a feature and V be a data value. Let $dom(s)$ denote the set of all propositions in $Tr(s) \cup Fa(s)$. Let $F_V{}'$ be a feature with the same components as F, with the exception of the following augmented definitions for the labeling functions and set of data propositions:*

- $Tr(s_0) = Tr(s_0) \cup \{p | p \notin dom(s_0) \wedge V(p) = \mathsf{true}\}$.
- $Fa(s_0) = Fa(s_0) \cup \{p | p \notin dom(s_0) \wedge V(p) = \mathsf{false}\}$.
- $D = D \cup \mathrm{domain}(V)$.

The next definition describes the semantics of a feature modelling a formula in a data environment.

Definition 10 *For a data environment D, a feature F, a CTL formula φ, and an initial state s_0 in F, s_0 models (does not model) φ in D iff*

[5] 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.

CONSTRAIN$(F_V, s_0, \varphi, \emptyset) = True(False)$ *for each* $V \in D$. *If neither of the above hold, the value of* φ *at* s_0 *in* D *is* $\perp$.

The following two definitions are needed for the statement of the theorems.

Definition 11 *For a data value* V *and a path environment* P*, define* $V \circ P$ *to be the path environment produced by augmenting* P *with the values of propositions in* V *that are not bound in* P.

Definition 12 *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* P *be a path environment from some initial state* s_0 *of* F_1 *to* s*. Let* c *be the result of* CONSTRAIN(F_1, φ, s, P)*. Define* c_R *to be* c *with every annotated formula* ψ_{s_t} *replaced with the value of* ψ *(*true, false*, or* $\perp$*) in the initial state of* F_2 *with which* s_t *connects (in the data environment of* ${F_1}_V{}'$*).*

The first theorem outlines the correctness of using CONSTRAIN compositionally on two features.

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)*.* ${F_1}_V{}' \circ F_2, s \models \varphi$ *if* V *satisfies* c_R*.*

Proof: Appendix B.

Using the above theorem, we can now prove soundness of VERIFY.

Theorem 2 *Let* $P, ENVS, Constraints, SUBS, \varphi$ *be as in the definition of* VERIFY*,* P' *be* P *augmented with the empty data environment (all data propositions of the product set to false). Let* φ_1 *be a subformula of* φ*. Let* F_i *be a feature in* P*, and* s_0 *an initial state of* F_i*. Then:*

$F_i \circ F_{i+1} \circ \ldots \circ B, s_0 \models \varphi_1$ *in the data environment of* $B \circ F_1 \circ \ldots \circ F_{i-1}$ *if* $SUBS[s_0, \varphi_1] = True$.

$F_i \circ F_{i+1} \circ \ldots \circ B, s_0 \not\models \varphi_1$ *in the data environment of* $B \circ F_1 \circ \ldots \circ F_{i-1}$ *if* $SUBS[s_0, \varphi_1] = False$.

Proof:

We will prove this by induction on the position of F_i in P.

Base case: F_i is the terminal base.

In this case the constraint for φ_1 has no temporal element. Corollary 2.1 and Theorem 1 give us the desired result.

Inductive step: Assume that the theorem holds to the right of F_i in the composition. Show that it holds for F_i.

By the inductive hypopaper and Corollary 2.1, $SUBS$ contains the correct value of each formula for F_{i+1}, since F_{i+1} is to the right of F_i. Therefore, Corollary 2.1 and Theorem 1 again give us the desired result.

5.4 An Alternative Verification Model

The methodology proposed in Section 5.3.2 follows a *client-side* strategy: the client performs all of the constraint generation and checking at assembly time. In that approach, a component producer may supply the features, but the client performs the verification after choosing which features to integrate; the verification algorithm therefore can exploit knowledge about all of the features that will be in the final product.

A different methodology would support *server-side* verification, where the feature producer would generate constraints, leaving the client to check compatibility at assembly time.[6] This framework parameterizes the verification algorithm over the features to be composed. This necessarily defers some model checking to assembly time, but all model checking remains local to individual features.[7]

Our constraint and data environment generation techniques support compositional verification under both approaches. The first style generates a large collection of constraints during analysis of individual features, but requires no model checking at composition time. The generated constraints are reusable across subsets and permutations of features within the product, thus amortizing the cost of product assembly over many products within the product family. The second style supports an algorithm in which the set of available features may be undecided until composition time; this flexibility leads to the potential additional model checking runs. In both cases, our approach is sound under compositional reasoning.

6 Experimental Validation

We have implemented the methodology described in this paper and tested it on a more extensive version of Hall's email case study [14]. Our implementation uses a restricted data environment generation algorithm that does not handle features that set data propositions within cycles; this restriction yielded a faster algorithm that was sufficient for the email case study.

As this paper attempts to improve on the methodology from the work of Li, Fisler and Krishnamurthi [18], our primary goal was to reproduce the results from that work on the same case study. The new methodology matches the results of the old one: on each property for which the prior approach reported no property violation, the new one reports that property as holding in the full system (this is a stronger claim than in the old methodology). Furthermore, on each property for which the prior approach reported a feature interaction, the new one also reported a potential interaction. As our primary goal was to confirm soundness of the approach, we do not report performance results.

7 Perspective and Future Work

Previous work [18] has argued that model checking did not seem an appropriate approach for feature verification, due to the clumsiness that arose when using 3-valued model checking to handle open features. This methodology is more satisfying. While this approach is still subject to the usual complexities and concerns underlying model checking, it does not

[6] Previous work [12, 18] has taken this approach.

[7] This model treats *C-external* slightly differently to preserve compositionality.

include undue *additional* complexities to achieve compositionality under openness.

The present methodology does, however, make some simplifying assumptions that we are working to relax as part of our future work. First, the methodology assumes that the compositions of features are acyclic. Features may contain cycles internally, but the graph showing connections between features must form a DAG; the algorithms presented in this paper assume feature connections form a linear chain rather than a DAG for simplicity. This is less of a restriction than it seems, because feature composition often resembles simple pipe-and-filter systems in which the only cycles lie within the base product (which we already support).

The second assumption concerns cycles within features. The current CONSTRAIN algorithm assumes that no cycle within a feature sets the value of a data proposition. We could support such assignments to local propositions (such as a loop counter), but not to propositions that must appear in the constraint. Although the examples we have used to test our methodology are consistent with this assumption, relaxing this remains an issue for future work.

8 Conclusion

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. This paper exploits a key insight about the nature of openness in compositional feature verification: features are open with regards to two distinct kinds of information. Specifically, 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 separate interface information for each form of openness. Using standard flow analysis techniques, we derive a propositional formula summarizing the data values that each feature provides to subsequent features; using CTL 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. The approach is compositional because model checking occurs only at the level of individual features during constraint generation.

This work stands in contrast to prior work on this problem, which modeled unavailable information using 3-valued logic. The corresponding methodology therefore required model checking algorithms for and produced interface formulas in this more complex logic. The resulting algorithm, while effective for detecting property violations, required multiple model checking passes per property per feature and yielded overly complex interfaces. The new methodology presented in this paper eliminates these problems without diminishing the effectiveness of compositional feature verification; in particular, it uses conventional 2-valued model checking, restricting 3-valued reasoning to purely propositional cases. We have implemented the new methodology and matched the results of a case study performed using the 3-valued methodology.

A Proof of Lemma 1

Lemma 1 Let F_1 and F_2 be features with data environments E_1 and E_2, respectively. Let R be the set of all transitions from states of F_1 to states of F_2 in the composition $F_1 \circ F_2$. Let $E'_{s_{t_2}}$ be the composed data environment for some s_{t_2} under R and $E_{s_{t_2}}$ be the data environment for s_{t_2} in the composition. $\langle s_{0_1}, DV \rangle \in E_{s_{t_2}}$ iff $\langle s_{0_1}, DV \rangle \in E'_{s_{t_2}}$.

Proof:

Let V be a data value, $\langle s_{0_1}, DV \rangle \in E_{s_{t_2}}$ and $\langle s_{0_1}, DV' \rangle \in E'_{s_{t_2}}$. We show that $V \in DV'$ iff $V \in DV$.

If: V is the data value for some $\Pi = s_{0_1}, s_1, ..., s_{t_1}, s_{0_2}, ..., s_{t_2}$, where $(s_{t_1}, s_{0_2}) \in R$. Note that if we partition Π at any arbitrary state to break it into two paths Π_1 and Π_2, then V is equivalent to $V_2 \cup \{\langle p, v \rangle \in V_1 | p \notin dom(V_2\}$, where V_1, V_2 are the data environments for Π_1 and Π_2 respecitively.

In particular, V is equivalent to $V_2 \cup \{\langle p, v \rangle \in V_1 | p \notin dom(V_2\}$ where V_1 is the data value for $s_{0_1}, ..., s_{t_1}$ and V_2 is the data value for $s_{0_2}, ..., s_{t_2}$. Therefore $V \in DV'$.

Only if: V is composed of some V_1 and V_2 from the data environments of F_1 and F_2 respectively. V_1 is the data value for some $\Pi_1 = s_{0_1}, s_1, ..., s_{t_1}$, and V_2 is the data value for some $\Pi_2 = s_{0_2}, ..., s_{t_2}$, where $(s_{t_1}, s_{0_2}) \in R$. V is the data value for $\Pi = \Pi_1, s_{t_1}, s_{0_2}, \Pi_2$, since V contains the last value of each proposition bound along that path. Therefore $V \in \langle s_{0_1}, DV \rangle$.

B Proof of Theorem 1

To prove this theorem, we will need two lemmas showing that PATH-CONSTRAIN operates correctly for EG and EU. There is simultaneous induction here: the lemmas assume that the theorem holds for the subformulas of EG, EU, and the theorem assumes that the lemmas hold for subformulas in the structural induction. The base case of the simultaneous induction is the propositional case in the theorem. We are showing that for each different type of formula, the result of getting the constraint on that formula in F_1, replacing temporal constraints with their values in F_2, and checking truth/falsehood in the data environment V is the same as that of running constrain on the composition augmented with the data environment V. Note also that we have defined c_R just before the statement of the constrain theorem in the paper. Finally, this is the underlying reasoning about replacing temporal subconstraints with their values: the values from F2 are the values of modelling the formula in the data environment of F1 o V. This means that constrain holds (doesn't hold) for all data values from F1 o V, which in particular means that it holds (doesn't hold) for whatever path value we are considering, since the path value is a data value.

Lemma 3 *Let F_1 and F_2 be features, s be a state in F_1, and φ_1 a CTL formula for which Theorem 1 holds. Let V be a data value coming into F_1. Let P be a path environment from some initial state s_0 of F_1 to s and let $marked \subset S_{F_1}$. Let c be the result of* PATH-CONSTRAIN$(F_1, EG\ \varphi_1, s, P, marked)$.

Then:

PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{true}$ *if V satisfies c_R.*

PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{false}$ *if V does not satisfy c_R.*

Proof:

We will prove this by induction on the number of states in $S_{F_1} \setminus marked$.

Base case: $S_{F_1} \setminus marked$ is empty.

In this case, $s \in marked$, meaning that $c_R = true$, and PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{true}$ as well (by the algorithm).

Inductive step: Assume that the lemma holds when $S_{F_1} \setminus marked$ has n states. Show that it holds when $S_{F_1} \setminus marked$ has $n+1$ states.

V satisfies c_R:

If $s \in marked$, then we can prove that the lemma holds using the same reasoning as the base case. If s is a terminal state, then V satisfies CONSTRAIN$(\varphi_1, s, path - env)_R$ and the successor of s models $EG\ \varphi_1$ in the data environment ${F_1}_V{}'$ (by the algorithm and the definition of c_R). Therefore, the lemma holds (since modelling EG in the data environment implies that constrain is true for all path environments–see reasoning for EX case of the theorem).

Therefore, assume that s is not a terminal state and $s \notin marked$. In this case,
$V \in sat($CONSTRAIN$(\varphi_1, s, path - env)_R)$ and for some successor t of s,
$V \in sat($PATH-CONSTRAIN$(EG\ \varphi_1, t, extend{-}path{-}env(t, path{-}env), marked \cup s)_R)$. Thus, by the assumptions of the lemma and the inductive hypopaper, PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{true}$.

V does not satisfy c_R:

If s is a terminal state, $V \notin sat($CONSTRAIN$(\varphi_1, s, path - env)_R)$ or the successor of s does not model $EG\ \varphi_1$ in the data environment ${F_1}_V{}'$. Therefore, the lemma holds (by the assumptions of the lemma and the definition of modelling in a data environment). s cannot be in $marked$ since c_R would then be a tautology.

Therefore, assume that s is not a terminal state and $s \notin marked$. In this case,
$V \notin sat($CONSTRAIN$(\varphi_1, s, path - env)_R)$ or for all successors t of s,
$V \notin sat($PATH-CONSTRAIN$(EG\ \varphi_1, t, extend{-}path{-}env(t, path{-}env), marked \cup s)_R)$. Thus, by the assumptions of the lemma and the inductive hypopaper, PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{false}$.

Lemma 4 *Let F_1 and F_2 be features, s be a state in F_1, and φ_1, φ_2 CTL formulas for which Theorem 1 holds. Let V be a data value coming into F_1. Let P be a path environment from some initial state s_0 of F_1 to s and $marked \subset S_{F_1}$. Let c be the result of* PATH-CONSTRAIN$(F_1, E\ \varphi_1 U\ \varphi_2, s, P, marked)$.

Then:

PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, E\ \varphi_1 U\ \varphi_2, s, V \circ P, marked) = \mathsf{true}$ *if* V *satisfies* c_R.

PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, E\ \varphi_1 U\ \varphi_2, s, V \circ P, marked) = \mathsf{false}$ *if* V *does not satisfy* c_R.

Proof: We will prove this by induction on the number of states in $S_{F_1} \setminus marked$.

Base case: $S_{F_1} \setminus marked$ is empty.

In this case, $s \in marked$, meaning that $c_R = false$, and PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, E\ \varphi_1 U\ \varphi_2, s, V \circ P, marked) = \mathsf{false}$ as well (by the algorithm).

Inductive step: Assume that the lemma holds when $S_{F_1} \setminus marked$ has n states. Show that it holds when $S_{F_1} \setminus marked$ has $n+1$ states.

V satisfies c_R:

s cannot be in $marked$ since c_R would then be unsatisfiable (by the algorithm). If s is a terminal state, then we are done using similar reasoning to Lemma 3 (with the addition that the satisfaction could come from φ_2–see reasoning below). Therefore, assume that s is not a terminal state and $s \notin marked$. In this case, we either have that $V \in sat(\text{CONSTRAIN}(\varphi_2, s, path{-}env)_R)$ or that $V \in sat(\text{CONSTRAIN}(\varphi_1, s, path{-}env)_R)$ and for some successor t of s, $V \in sat(\text{PATH-CONSTRAIN}(E\ \varphi_1 U\ \varphi_2, t, extend{-}path-env(t, path-env), marked \cup s)_R)$. In the former, the assumptions of the lemma show that the lemma holds. In the latter, by the assumptions of the lemma and the inductive hypopaper, PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{true}$.

V does not satisfy c_R:

If s is in marked, then we prove that the lemma holds using the same reasoning as the base case. If s is a terminal state, then we are done using similar reasoning to Lemma 3. Therefore, assume that s is not a terminal state and $s \notin marked$. In this case, we have that $V \notin sat(\text{CONSTRAIN}(\varphi_2, s, path{-}env)_R)$, and that $V \notin sat(\text{CONSTRAIN}(\varphi_1, s, path{-}env)_R)$ or for all successors t of s, $V \notin sat(\text{PATH-CONSTRAIN}(E\ \varphi_1 U\ \varphi_2, t, extend{-}path-env(t, path-env), marked \cup s)_R)$. Thus, by the assumptions of the lemma and the inductive hypopaper, PATH-CONSTRAIN$({F_1}_V{}' \circ F_2, EG\ \varphi_1, s, V \circ P, marked) = \mathsf{false}$.

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 P be a path environment from some initial state s_0 of F_1 to s. Let c be the result of CONSTRAIN(F_1, φ, s, P).

Then:

CONSTRAIN$({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{true}$ if V satisfies c_R.

CONSTRAIN$({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{false}$ if V does not satisfy c_R.

Proof:

Unless otherwise noted, CONSTRAIN is operating within the context of F_1. Also note that for every formula for which we assume that Theorem 1 holds, we also assume that Lemma 3 and 4 hold–this is the simultaneous induction.

Case $\varphi = p$ (an atomic proposition):

$V \in sat(c_R)$:

There are three cases: either p is a control proposition of F_1 and $p \in Tr(s)$, p is a data proposition and $P(p) = T$, or p is a data proposition that is not bound by P and $V(p) = T$. In each case, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, p, s, V \circ P) = \mathsf{true}$.

$V \notin sat(c_R)$:

There are four cases: either p is a control proposition of F_1 and $p \in Fa(s)$, $p \in C - EXTERN$, p is a data proposition and $P(p) = F$, or p is a data proposition that is not bound by P and $V(p) = F$. In each case, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, p, s, V \circ P) = \mathsf{false}$.

Case $\varphi = \neg\varphi_1$:

We assume that the theorem holds for φ_1.

$V \in sat(c_R)$: $V \notin sat((\text{CONSTRAIN}(\varphi_1, s, P)_R)$. By our assumption this means that $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, s, V \circ P) = \mathsf{false}$. Thus, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{true}$.

$V \notin sat(c_R)$: $V \in sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$. By our assumption this means that $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, s, V \circ P) = \mathsf{true}$. Thus, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{false}$.

Case $\varphi = \varphi_1 \vee \varphi_2$:

We assume that the theorem holds for φ_1, φ_2.

$V \in sat(c_R)$: Either $V \in sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$ or $V \in sat(\text{CONSTRAIN}(\varphi_2, s, P)_R)$. WLOG assume that $V \in sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$. By our assumption this means that $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, s, V \circ P) = \mathsf{true}$. Therefore, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{true}$.

$V \notin sat(c_R)$: $V \notin sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$ and $V \notin sat(\text{CONSTRAIN}(\varphi_2, s, P)_R)$. By our assumption this means that $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, s, V \circ P) = \mathsf{false}$ and $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_2, s, V \circ P) = \mathsf{false}$. Therefore, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{false}$.

Case $\varphi = \varphi_1 \wedge \varphi_2$:

We assume that the theorem holds for φ_1, φ_2.

$V \in sat(c_R)$: $V \in sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$ and $V \in sat(\text{CONSTRAIN}(\varphi_2, s, P)_R)$. By our assumption this means that $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, s, V \circ P) = \mathsf{true}$ and $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_2, s, V \circ P) = \mathsf{true}$. Therefore, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{true}$.

$V \notin sat(c_R)$: Either $V \notin sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$ or $V \notin sat(\text{CONSTRAIN}(\varphi_2, s, P)_R)$. WLOG assume that $V \notin sat(\text{CONSTRAIN}(\varphi_1, s, P)_R)$. By our assumption this means that $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, s, V \circ P) = \mathsf{false}$. Therefore, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi, s, V \circ P) = \mathsf{false}$.

Case $f = EX\ \varphi_1$:

We assume that the theorem holds for φ_1. We break the proof into two cases: first, where s is a non-terminal state, and second, where s is a terminal state.

s is a non-terminal state:

$V \in sat(c_R)$: $V \in sat(\text{CONSTRAIN}(\varphi_1, t, extend - path - env(P, t))_R)$ for some $t \in succ(s)$. By our assumption, $\text{CONSTRAIN}({F_1}_V{}' \circ F_2, \varphi_1, t, V \circ$

$extend - path - env(P, t)) = \mathsf{true}$. Thus, $\textsc{constrain}({F_1}_V' \circ F_2, \varphi, s, V \circ P) = \mathsf{true}$ (by the algorithm).

$V \notin sat(c_R)$: $V \notin sat(\textsc{constrain}(\varphi_1, t, extend - path - env(P, t))_R)$ for all $t \in succ(s)$. By our assumption, $\textsc{constrain}({F_1}_V' \circ F_2, \varphi_1, t, V \circ extend{-}path{-}env(P, t)) = \mathsf{false}$ for all $t \in succ(s)$. Thus, $\textsc{constrain}({F_1}_V' \circ F_2, \varphi, s, V \circ P) = \mathsf{false}$ (by the algorithm).

s is a terminal state:

$V \in sat(c_R)$: This means that the state t_0 to which s is connected in the composition models φ_1 in the data environment of ${F_1}_V'$, which in particular implies that $\textsc{constrain}({F_1}_V' \circ F_2, \varphi_1, t_0, V \circ extend - path - env(P, t_0)) = \mathsf{true}$. Therefore, $\textsc{constrain}({F_1}_V' \circ F_2, \varphi, s, V \circ P) = \mathsf{true}$.

$V \in sat(c_R)$: This means that the state t_0 to which s is connected in the composition does not model φ_1 in the data environment of ${F_1}_V'$, which in particular implies that $\textsc{constrain}({F_1}_V' \circ F_2, \varphi_1, t_0, V \circ extend - path{-}env(P, t_0)) = \mathsf{false}$ (by the definition of modelling in a data environment and the fact that P is in the data environment of ${F_1}'_V$). Therefore, $\textsc{constrain}({F_1}_V' \circ F_2, \varphi, s, V \circ P) = \mathsf{false}$.

Case $f = EG\ \varphi_1$:

We assume that the theorem holds for φ_1. Lemma 3 then gives us the desired result.

Case $f = E\ \varphi_1\ U\ \varphi_2$:

We assume that the theorem holds for φ_1, φ_2. Lemma 4 then gives us the desired result.

References

[1] 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.

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

[3] Don Batory. Product-line architectures. In *Smalltalk and Java Conference*, October 1998.

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

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

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

[7] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology*, Accepted for publication 2003.

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

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

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

[11] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

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

[13] Arie Gurfinkel, Benet Devereux, and Marsha Chechik. Model exploration with temporal logic query checking. In *Symposium on the Foundations of Software Engineering*. ACM Press, 2002.

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

[15] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

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

[17] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Interfaces for modular feature verification. In *IEEE International Symposium on Automated Software Engineering*, pages 195–204. IEEE Press, 2002.

[18] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. In *Symposium on the Foundations of Software Engineering*, pages 89–98. ACM Press, 2002.

[19] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, March 1999.

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

[21] Christian 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.

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

[23] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, December 1999.