Feature Specification and Automatic Conflict Detection*

Amy P. Felty†
School of Information Technology and Engineering, University of Ottawa
afelty@site.uottawa.ca

Kedar S. Namjoshi
Bell Laboratories,
Lucent Technologies
kedar@research.bell-labs.com

Abstract. We present a formal feature specification language and a method of automatically detecting feature conflicts ("undesirable interactions") at the *specification* stage. Early conflict detection can help prevent costly and time-consuming problem fixes during implementation. Features are specified in linear temporal logic; two features conflict essentially if their specifications are mutually inconsistent under axioms about the underlying system behavior. We show how this inconsistency check may be performed automatically with existing model checking tools. The model checking tools can also be used to provide witness scenarios, both when two features conflict as well as when the features are mutually consistent. Both types of witnesses are useful for refining the specifications. We have implemented a conflict detection tool, FIX (Feature Interaction eXtractor), that uses the model-checker COSPAN for the inconsistency check. We describe our experience in applying this tool to a collection of feature specifications derived from the Telcordia (Bellcore) standards.

1 Introduction

Telecommunications services are typically marketed to customers as groups of *features* such as call-waiting and call-forwarding. Since the groups are flexible, an individual feature is usually specified without knowledge of which other features it may be grouped with. This facilitates modular design and implementation; however, as features in a group can be active concurrently, problems arise when the feature requirements mandate conflicting behavior. Individual implementations may resolve such conflicts in different ways, leading to unpredictable behavior in the system as a whole. It is therefore essential to detect and resolve such feature conflicts as early as possible, preferably in the specification stage itself [1].

With this motivation, we have developed a formal feature specification language, and a method of automatically detecting feature conflicts at the specification stage,

*In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, IOS Press, 2000.

†This work was done while the author was at Bell Laboratories.

[1]This notion of conflict roughly corresponds to *feature interference* or *service interference* as discussed in the literature (see [12], for example); in contrast, *interaction* is often used more generally and includes interactions that may be desirable.

which is implemented in a detection tool. Features are specified by describing their *temporal* behavior. For instance, a typical informal specification for call forwarding is that "If entity x has call forwarding enabled and calls to x are to be forwarded to z then, whenever x is busy, any incoming call from y to x is eventually forwarded to z". This informal description can be expressed precisely in our specification language, as described in Section 3. The language itself may be viewed as a sugared version of temporal logic or ω-automata. Specifying features as temporal formulae abstracts from specific state-machine implementations, allowing any implementation that satisfies the specifications.

Given that specifications are temporal formulae, the natural way to define a feature conflict is that the feature formulae are mutually inconsistent; i.e., their conjunction is unsatisfiable. As discussed in Section 4.1, to detect feature conflicts, we may also need to include *axioms* about the underlying system. The system axioms describe properties that should be true of *any* reasonable system implementation. The need for such system axioms in one form or another (for example, the *network properties* in [9]) has arisen in a variety of approaches to the feature interaction problem. In our case, typical axioms for telephony include the following: (i) the system should not disconnect an established call, and (ii) if a call attempt is rejected, no connection should be established until the next attempt. These axioms are also specified in the same specification language as the features. Specifying the system by axioms abstracts from particular implementations, resulting in conflict reports that have wider applicability.

Conflict detection is thus reduced to a satisfiability test for temporal formulae. By considering only a finite number of entities, the feature specifications can be made propositional, and the test can be performed automatically with a model checking tool. We have developed a tool, FIX (Feature Interaction eXtractor), that reads in feature specifications, converts them into ω-automata descriptions, and uses the model checking tool COSPAN [10] to perform the satisfiability test. The detection process is fully automated. The model checker provides *witness* computations for either outcome. If no conflict is detected, the witness describes a non-conflicting computation of the system; examining this computation often reveals assumptions about the system that need to be added as axioms. If a conflict is detected, the witness computation describes a particular scenario where the features conflict. By examining this scenario, one can determine either the proper resolution of the conflict, or whether the specifications need to be modified. Our specification method includes a mechanism that makes it easy to specify dynamic priorities (i.e., dependent on system state) between conflicting features. Our experience so far has been that this detection process is reasonably efficient and quite accurate; for the set of features to which we have applied this method, we have been able to detect most of the interactions given in the Telcordia (Bellcore) standards, as well as some new ones.

The rest of the paper is structured as follows. Section 2 contains a short background on temporal logic, ω-automata and model checking. We motivate and describe our specification language in Section 3. The precise definition of feature conflict and the detection method is described in Section 4. We have applied our tool to several Telcordia feature specifications; this is described in Section 5. The paper concludes with a discussion of related work and conclusions in Section 6.

2 Background

In this section, we provide a short background on linear temporal logic, ω-automata, and model checking.

2.1 Linear Temporal Logic

Linear-time temporal logic (usually abbreviated as *LTL*) was first suggested as a protocol specification language in [16]. Formulae in the logic define sets of *infinite* sequences; hence, the logic is particularly well suited to describe time dependent properties of concurrent, reactive systems such as telephony and other network protocols. Formally, *LTL* formulae are parameterized by a set of *atomic propositions*, AP, and are defined by the following syntax:

1. Every proposition P in AP is a formula,
2. For formulae f and g, $(f \wedge g)$ and $\neg(f)$ are formulae,
3. For formulae f and g, $\mathsf{X}(f)$ and $(f \mathbin{\mathsf{U}} g)$ are formulae.

The temporal operators are X (read as "next-time") and U (read as "until"). An infinite sequence of atomic proposition valuations can be defined as a function from $\mathbf{N}$ to 2^{AP}. We write $\sigma, i \models f$ to mean that the infinite sequence $\sigma : \mathbf{N} \to 2^{AP}$ *satisfies* the formula f at position i. The *language* of f, denoted by $\mathcal{L}(f)$, is the set $\{\sigma \mid \sigma, 0 \models f\}$. The satisfaction relation can be defined by induction on the structure of f as follows.

1. For a proposition P, $\sigma, i \models P$ iff $P \in \sigma(i)$,
2. $\sigma, i \models \neg(f)$ iff $\sigma, i \models f$ is false,
   $\sigma, i \models (f \wedge g)$ iff both $\sigma, i \models f$ and $\sigma, i \models g$ are true,
3. $\sigma, i \models \mathsf{X}(f)$ iff $\sigma, i+1 \models f$,
   $\sigma, i \models (f \mathbin{\mathsf{U}} g)$ iff there exists j, $j \geq i$, such that $\sigma, j \models g$ and for every k, $i \leq k < j$, $\sigma, k \models f$.

Other connectives can be defined in terms of these basic connectives: $(f \vee g)$ is $\neg(\neg f \wedge \neg g)$; $(f \Rightarrow g)$ is $\neg f \vee g$; $\mathsf{F}(g)$ ("eventually g") is $(\mathit{true} \mathbin{\mathsf{U}} g)$; $\mathsf{G}(f)$ ("always f") is $\neg\mathsf{F}(\neg f)$, and $(f \mathbin{\mathsf{W}} g)$ ("f holds unless g") is $(\mathsf{G}(f) \vee (f \mathbin{\mathsf{U}} g))$.

2.2 Automata on infinite sequences

Temporal properties can also be specified by finite-state automata that recognize *infinite* input sequences. Such automata are known as Büchi automata [4] or as ω-automata. A Büchi automaton $\mathcal{A}$ is specified by a tuple $(S, \Sigma, \Delta, I, F)$, where:

- S is a finite set of *states*,
- Σ is a finite set known as the *alphabet*,
- $\Delta \subseteq S \times \Sigma \times S$, is the *transition relation*,
- $I \subseteq S$ is the set of *initial* states,
- $F \subseteq S$ is the set of *accepting* states.

A *run* of $\mathcal{A}$ on an infinite sequence $\sigma : \mathbf{N} \to \Sigma$ is an infinite sequence $r : \mathbf{N} \to S$ of states such that: (i) $r(0) \in I$, and (ii) for each $i \in \mathbf{N}$, $(r(i), \sigma(i), r(i+1)) \in \Delta$. A run r is *accepting* iff one of the states in F appears *infinitely often* along r. The *language* of the automaton, $\mathcal{L}(\mathcal{A})$, is the set of infinite sequences on which $\mathcal{A}$ has an accepting run. Büchi automata (with $\Sigma = 2^{AP}$) are strictly more powerful than linear temporal logic at defining sets of sequences. There is a (worst-case exponential) translation from *LTL* formulae to equivalent Büchi automata; see [18] for a survey.

2.3 Model Checking

A program generates a set of computation sequences. For reactive programs where *non-termination* is desirable, such as operating systems and telephony protocols, the sequences are infinite, in general; hence, temporal logic or Büchi automata can be used to describe properties of the programs. For instance, mutual exclusion may be written as $\mathsf{G}(\neg(\mathit{Critical}_0 \wedge \mathit{Critical}_1))$, and eventual access as $\mathsf{G}(\mathit{Waiting} \Rightarrow (\mathit{Waiting}\ \mathsf{U}\ \mathit{Granted}))$.

For programs with finitely many states, a fully automated procedure known as Model Checking [5, 17] can be used to determine if a property holds of all computations of the program. A finite state program can be represented by a Büchi automaton with the trivial acceptance condition $F = S$; hence, model checking becomes the language containment question $\mathcal{L}(\mathit{Program}) \subseteq \mathcal{L}(\mathit{Property})$ [19].

Model Checking tools based on language containment include COSPAN [10] and VIS [3]. If the specification fails to hold of the program, the tool generates a computation that is a *witness* to this failure; i.e., one that is in the set $\mathcal{L}(\mathit{Program}) \cap \overline{\mathcal{L}(\mathit{Property})}$. We make use of this capability in our conflict detection method (Section 4).

3 Feature Specification

In this section, we describe and define our feature specification language and the methodology we have used to set up the feature conflict check. The details of this check are presented in the following section.

In order to specify features, we have to begin with some informal understanding of the term "feature". In the rest of the paper, we restrict ourselves to telephony features; however, our specification language and the conflict detection algorithm can also be applied to specifications of features in other kinds of systems.

A telephony feature, such as call waiting or call forwarding, typically specifies the behavior, over time, of one or more entities in terms of their current *state* and a set of input *events*. The informal specification given earlier for call forwarding is an example: "If entity x has call forwarding enabled and calls to x are to be forwarded to z then, whenever x is busy, any incoming call from y to x is eventually forwarded to z". In this specification, we can distinguish several *predicates* that describe the state of entity x: $\mathit{call\_forwarding\_enabled}(x)$, $\mathit{forward\_from\_to}(x, z)$, $\mathit{forwarded\_call\_from\_to}(y, x, z)$, $\mathit{busy}(x)$, and the predicate $\mathit{incoming\_call\_from\_to}(y, x)$ that describes the occurrence of an event. The rest of the sentence uses boolean operators and temporal operators (i.e., "whenever", "eventually"). Hence, we believe that a particularly appropriate way of specifying a feature is by a collection of temporal formulae (or automata) that are defined over a set of predicates that denote states or events of the system.

The specification notation that we have developed is a sugared version of *LTL*. Each feature is specified in a separate file; for instance, call forwarding is specified in the file "call_forwarding.spec". Each specification consists of definitions of basic and derived predicates, and a list of properties. We use the symbols $+, \&, \sim, =>$ to denote the boolean operators $\lor$, $\land$, $\neg$, $\Rightarrow$ respectively.

The properties are defined in terms of predicates that indicate relationships between entities in the system. There are two pre-defined predicates: $eq(x, y)$, which denotes equality of the entities x and y and, for each feature, a predicate $disable(x)$, which indicates that the feature specification is to be disabled at entity x. The latter predicates are used for selectively disabling features in order to resolve conflicts. The identifiers x, y etc. are *variables* which can be instantiated by constants representing entities in the system. We allow existential quantification over entities. We use it, for example, to specify predicates such as $is\_on\_hold(x) = (exists\ y : has\_on\_hold(y, x))$. A restricted form of existential quantification represents quantified variables by "_"; for instance, the above definition may also be written as $is\_on\_hold(x) = has\_on\_hold(\_, x)$. The scope of an existential quantifier in such an abbreviated form includes only the predicate containing the "_" symbol.

The general form of a property specification is shown below. The symbols $e0$, $p0$, $e1$, $p1, \ldots, eN$, p, r, d are boolean expressions formed out of the basic predicates. The keyword `until` may be replaced with the keyword `unless` to define a weaker specification.

```
property <Name>
{
 event: e0   persists: p0
 event: e1   persists: p1
 ...
 event: eN
-----------------------
 persists: p until: r discharge: d
}
```

The event and persists conditions above the dashed line indicate the *precondition* of the property; the persists-until-discharge triple (or a persists-unless-discharge triple) indicates the *postcondition* of the property. Informally, the property states that "whenever the precondition holds, the postcondition holds subsequently".

The precondition has the following informal reading: "$e0$ holds, followed by a period where $(p0 \land \neg e1)$ is true, then $e1$ holds, followed by a period where $(p1 \land \neg e2)$ is true, etc., until eN holds." In extended regular expression notation, this can be written succinctly as $e0; (p0 \land \neg e1)^*; e1; (p1 \land \neg e2)^*; \ldots; eN$. We say that a property is *enabled* at a point on a computation iff its precondition is true of a prefix that ends at the point.

The postcondition should hold at every point on a computation where the property is enabled. The "persists: p until: r discharge: d" notation translates to the *LTL* formula $(p \ \mathsf{U}\ (r \lor d))$; with `unless` in place of `until`, it corresponds to the *LTL* formula $(p \ \mathsf{W}\ (r \lor d))$. While the `discharge` condition may seem technically unnecessary, it makes a distinction that is important for the specifier. The `until` condition is thought of as specifying the *desired* outcome, while the `discharge` condition is thought of as specifying the *exception* conditions that cause the property to be trivially satisfied. We make use of this distinction in our conflict test. Any of the three components of the postcondition can be omitted; the choice between `until` and `unless` defaults to `unless`, the `persists` condition defaults to *true*, and the `unless` and `discharge` conditions default to *false*.

The easiest way to define the complete property in *LTL* is to consider its negation: the property is false of an infinite sequence iff there is a point where the precondition holds but the postcondition fails to hold. To illustrate the translation, consider the property below.

```
event:e0 persists:p0 event:e1
-----------------------------
persists:p until:r discharge:d
```

The *LTL* property $\neg\mathsf{F}(e0 \ \wedge \ \mathsf{X}((p0 \ \wedge \ \neg e1) \ \mathsf{U} \ (e1 \ \wedge \ \neg(p \ \mathsf{U} \ (r \ \vee \ d)))))$ is equivalent to this specification. The general case can be handled in a similar manner, increasing the depth of nesting for successive event-persists pairs. This translation indicates why it is better to use a sugared notation than to use *LTL* directly. We consider such a formula with free variables $x, y, \ldots$ to represent the infinite family of propositional *LTL* formulae defined by instantiating the free variables with constants. We use such instantiations in our conflict test, but the presence of free variables makes it simple to consider alternative bindings of constants to variables.

We have shown how features may be represented by formulae in *LTL* over a set of predicates. The predicates are, however, not independent – any underlying telephony system imposes some constraints between the predicates. For instance, $busy\_tone(x)$ and $call\_waiting\_tone(x)$ are mutually exclusive. Constraints such as these can be considered as an *axiomatization* of the switching infrastructure of a telephony system. In the specification language, constraints are specified using the same syntax as properties, except that the form begins with the keyword `constraint` instead of `property`.

This approach of casting the entire specification as a collection of temporal logic formulae differs from the common method of constructing state machine models of the switching system and the individual entities. State machine models fix a particular implementation – however abstract – which can create feature conflicts that may be avoided in other implementations. In addition, modifying a state machine to change or add properties is quite difficult, while with temporal logic this can be done simply by changing or adding to the property specification. We believe that this considerably simplifies the maintainance of the specification. While state machines can sometimes be more succinct at representing a collection of closely related properties, the benefits of adopting a formula-based approach outweigh this disadvantage.

4 Feature Conflict Detection

Given that a feature is specified as a temporal logic formula, how can we define "conflict" (i.e., an "undesirable interaction")? We motivate our current definition through an analysis of successively stronger formulations. We then describe our detection method and analyze its strengths and weaknesses. In the following, it should be understood that we are referring to specific instantiations of the features (i.e., binding the free variables with constants). This is indicated by using the letters $a, b, \ldots$ instead of $x, y, \ldots$ in the formulae. We say that a feature is enabled if one of the properties of the feature is enabled.

4.1 Formulating "Conflict" Precisely

Consider the following definition of feature conflict: features A and B conflict iff there *does not* exist a system where every computation satisfies both the specifications A and B. We can form a simpler, equivalent formulation by applying the following general theorem.

Theorem 1 *For any propositional LTL formulae f and g, there exists a system that satisfies f on some computation and satisfies g on all computations if and only if the formula $f \wedge g$ is satisfiable.*
Proof Sketch. In the left-to-right direction, consider the computation of the witness system that satisfies f. As g is true of all computations, it must also satisfy g; hence, $f \wedge g$ is satisfiable. In the other direction, if $f \wedge g$ is satisfiable, there exists a path ending in a cycle that satisfies both formulae (see [18] for details). This path defines a system with the required properties.
End Proof.

Instantiating the theorem with f as "*true*" and g as "Spec_A and Spec_B", we get that the feature conflict definition above is equivalent to the following one.

Definition 1 *Features A and B conflict iff the formula $(\mathit{Spec}_A \wedge \mathit{Spec}_B)$ is unsatisfiable; i.e., in every computation, some feature property does not hold.*

This definition, however, turns out to be inadequate. Consider the two features A and B defined by $\mathit{Spec}_A = \mathsf{G}(\mathit{calls}(a,b) \Rightarrow \mathsf{F}(\mathit{connected}(a,b) \vee \mathit{disconnect}(a)))$ ("Whenever a calls b, eventually a and b are connected, if a does not disconnect"), and $\mathit{Spec}_B = \mathsf{G}(\mathit{calls}(a,b) \Rightarrow \mathsf{F}(\mathit{forwards}(a,b,c) \vee \mathit{disconnect}(a)))$ ("Whenever a calls b, the call is eventually forwarded to c, if a does not disconnect").

Informally, these specifications are conflicting, since forwarding from b and connecting to b should not both happen for a single call. Yet the conjunction of the formulae is satisfiable: consider the computation in which $\mathit{calls}(a,b)$ is always false! The problem here is that it is always possible to satisfy a feature specification in a system where the feature is always disabled. Hence, we would like to consider only those systems for which there exist computations where both features can be enabled together. We choose to consider only computations where both features are enabled together infinitely often – a computation where the features are enabled together once, but disabled forever from some point on is, in a sense, artificially restricted. Instantiating Theorem 1 with f as "infinitely often A and B enabled" and g as "Spec_A and Spec_B", we are led to our second formulation.

Definition 2 *Features A and B conflict iff the two features can be enabled together infinitely often, but in every such computation, some feature property does not hold.*

Even with the strengthened definition, the two features in our example are still non-conflicting! Consider the computation in which whenever $\mathit{calls}(a, b)$ is true, eventually $\mathit{connected}(a, b)$ holds, followed by $\mathit{forwards}(a, b, c)$. The problem here is that we have failed to account for the constraint that prevents the same call being both connected and forwarded. This is not a feature property; it should be part of the system axioms. We would like to constrain the possible implementations further so that they satisfy these axioms along all computations. Instantiating Theorem 1 with f as "infinitely often A and B enabled" and g as "system axioms and Spec_A and Spec_B", we are led to our third formulation.

Definition 3 *Features A and B conflict iff the two features can be enabled together infinitely often under the system axioms, but in every computation where the features are enabled together infinitely often and the system axioms also hold, some feature property does not hold.*

It is still true that the example features are non-conflicting! Consider the computation in which after $\mathit{calls}(a, b)$ holds, $\mathit{disconnect}(a)$ is true before either $\mathit{connected}(a, b)$ or $\mathit{forwards}(a, b, c)$ holds. Both specifications are thus satisfied trivially. It is for such a situation that we make use of the distinction between `until/unless` and `discharge` conditions. We would like to rule out those computations where discharge events occur while the feature is *pending*, i.e., enabled but not satisfied. Adding this property to the previous instantiation of g and applying Theorem 1, we get the following final definition of feature conflict.

Definition 4 (Feature Conflict) *Features A and B conflict iff A and B can be enabled together infinitely often under the system axioms, and for every computation where*

1. *The system axioms hold, and*
2. *A and B are enabled together infinitely often, and*
3. *The discharge condition for a feature does not occur while the feature is pending,*

some feature property does not hold.

Conditions 2 and 3 can be expressed with simple formulae of temporal logic. For instance, "p holds infinitely often" is expressed by $\mathsf{GXF}(p)$ and "d does not occur between occurrences of p and q" is expressed by $\mathsf{G}(p \Rightarrow (\neg d \,\mathsf{W}\, q))$.

4.2 Automatic Detection

Each conflict test is performed on a specific instantiation of the features. The parameterized form of the feature specification makes it easy to instantiate different configurations – for instance, one where entity a has call-forwarding and entity b has call-waiting. In general, two *LTL* properties f and g are inconsistent iff $\mathcal{L}(f) \cap \mathcal{L}(g) = \emptyset$, which is true iff $\mathcal{L}(f) \subseteq \overline{\mathcal{L}(g)}$. This is exactly the model checking question with f

as the program and $\neg g$ as the property. Hence, a model checker can be used to detect feature conflicts. For features A and B, system axioms C, and auxiliary automata D that specify conditions 2 and 3 of Definition 4, the inconsistency check can be written as $\mathcal{L}(A) \cap \mathcal{L}(B) \cap \mathcal{L}(C) \cap \mathcal{L}(D) = \emptyset$, which is equivalent to $\mathcal{L}(C) \cap \mathcal{L}(D) \subseteq \overline{\mathcal{L}(A)} \cup \overline{\mathcal{L}(B)}$. This is the form used in our implementation.

We have developed a tool called FIX (for Feature Interaction eXtractor) that uses the model checker COSPAN [10] for the conflict check. In COSPAN, both properties and constraints are represented by ω-automata. FIX translates the constraints C and the feature specifications A, B into COSPAN automata that accept the specified languages. Each feature is translated to a parameterized automaton which is instantiated as needed for each particular test. Since the automata representing conditions 2 and 3 of the definition are independent of the particular features, they are obtained from a library and instantiated on each use with the enabling condition of the particular features.

The model checker declares failure if the set inclusion above is false; i.e., if the properties *do not* conflict. The non-conflict may be due to weak system axioms, or (rarely) because the instantiation defines a system without enough entities to exhibit a conflict. Since the model checker declares failure, it produces a witness computation for which the axioms and both features hold. Inspection of this witness computation often reveals constraints that need to be included in the system axioms. Even if this is not the case, a "no conflict" report should be, in general, considered inconclusive, as the check is performed for a particular system configuration (i.e., a fixed number of entities).

On the other hand, a "conflict" result is conclusive; but, as the model checker declares success, no witness is produced for the conflict. To produce a witness, we perform another check: $\mathcal{L}(C) \cap \mathcal{L}(D) \cap \mathcal{L}(A) \subseteq \mathcal{L}(B)$. As there is a conflict, this check must fail, so the model checker produces a computation that satisfies C, D and A but does not satisfy B. This computation describes a scenario in which both features are enabled together infinitely often and A holds, but B does not hold.

5 Case Study

We have applied our tool to a collection of feature specifications derived from the Telcordia standards. We report on the results for ten of these features, each checked against the nine others. One of the features we consider is Anonymous Call Rejection (ACR). Calls to a subscriber having this feature will not go through when the caller prevents her number from being displayed on the subscriber's caller ID device. The following property is one example from the 6 properties which specify this feature.

```
property ACR_Normal_Operation_3
{
  event: ACR(x) & call_req(x,y) & ~DN_allowed(y) & resources_for_ACR_annc(x)
  -----------------------
  persists: call_req(x,y)
  until: ACR_annc(y,x)
  discharge: onhook(y)
}
```

Informally, it states that if x subscribes to ACR and if there is a call request to x from y, and if furthermore the presentation of y's number is restricted and resources for the ACR denial announcement are available, this should cause y to receive the ACR announcement, unless y gives up and goes back on hook first. Note that *call_req* occurs both as an event and a persisting condition. In our model, events are not a primitive concept; they are points in time in which a formula becomes true. For example, $\mathit{call\_req}(x, y)$ becomes true at some point after completion of dialing and continues to hold until there is some resolution of the call such as a connection or an announcement.

A second feature that we consider is Call Forwarding Busy Line (CFBL), where the subscriber gives a number to which all calls will be forwarded when the subscriber's line is busy. The following is one of 3 properties specifying this feature.

```
property CFBL_Normal_Operation_1
{
  event: CFBL(x) & ~idle(x) & ~forwarding(x,_,z) &
         same_switch(x,z) & le_five_forwards(y) & call_req(x,y)
  -----------------------
  persists: call_req(x,y)
  until: forwarding(x,y,z)
  discharge: onhook(y)
}
```

This property states that if (1) x subscribes to CFBL, (2) x is not idle, (3) all previously forwarded calls from x to z have terminated, (4) x and z are on the same switch, (5) the incoming call from y has been forwarded at most 5 times and (6) there is an incoming call from y, then the incoming call from y to x will be forwarded to z, unless y goes back on hook in the meantime.

These two properties provide one example of the kind of conflict that may arise. Consider the case when x and y in the ACR property are instantiated with a and b, respectively and x, y, z of the CFBL property are instantiated with a, b, and c, respectively. Furthermore, suppose that all of the predicates in both events hold simultaneously. Thus a subscribes to both ACR and CFBL and has an incoming call from b. The two features require that the incoming call be resolved in different ways: ACR requires that b receive the ACR denial announcement, while CFBL requires that the call be forwarded to c. The information required from the system axioms in order for this conflict to be detected by our tool is that (1) a call request is distinct from a call resolution and (2) that the two resolutions cannot occur at the same time. These properties are expressed by the following constraints.

```
constraint call_req_not_resolution
{
event: true
 ---------------
persists: ~(call_req(x,y) & (ACR_annc(y,x) + forwarding(x,y,_)))
}

constraint distinct_resolutions
{
event: true
```

Table 1: Features, Number of Properties used in Specification, and Descriptions

ACR	Anonymous Call Rejection	6	Allows subscriber to reject calls from parties who have a privacy feature that prevents the delivery of their calling number to the called party. When active, the call is routed to a denial announcement and terminated.
CFBL	Call Forwarding Busy Line	3	A telephone-company-activated feature that forwards incoming calls to a subscriber to another line when the subscriber is busy.
CFDA	Call Forwarding Don't Answer	4	Incoming calls to the subscriber are forwarded when the subscriber doesn't answer after a specified time interval.
CFMB	Call Forwarding Make Busy	1	Allows subscriber to press a key to put phone into a busy state so that all calls will be forwarded.
CFV	Call Forwarding Variable	7	Allows subscriber to specify a number to which all calls will be forwarded.
CW	Call Waiting	16	Informs a busy subscriber that another call is waiting by playing a tone. The subscriber may flash, placing the original call on hold and answer the new call, or may go on-hook, in which case the subscriber is rung and connected to the new call upon answer.
DOS	Denied Originating Service	2	Provides the capability to deny a subscriber from making calls.
DTS	Denied Terminating Service	2	Provides the capability to deny terminating calls to a subscriber.
PKUP	Call Pickup	2	Allows one station to answer a call directed to another station within a business group.
RDA	Residential Distinctive Alerting	2	Allows the subscriber to designate special telephone numbers that may be identified using distinctive alerting treatment.

```
 ---------------
persists: ~(forwarding(x,y,_) & ACR_annc(y,x))
}
```

The first property states that at any point in time when x has an outstanding call request to y, y is neither receiving the ACR denial announcement from x nor having its call to x forwarded. The second property states that a call to x from y is not being forwarded at the same time that y is receiving the ACR denial announcement. Without these constraints there would be no conflict. For example, without the second constraint, nothing prevents the call from being forwarded at the same time that the caller is given an announcement. The conflict in this case should be resolved by giving precedence to the ACR feature; the CFBL property should only be required to hold when the subscriber does not also subscribe to ACR.

Table 1 describes the 10 features we consider here. Their names, descriptions, and number of properties in each of their specifications are given in the table. Table 2

Table 2: Number of Conflicting Property Pairs for each Pair of Feature Specifications

	CFBL	CFDA	CFMB	CFV	CW	DOS	DTS	PKUP	RDA
ACR	8	5	4	3	8	2	4	4	0
CFBL	—	0	2	2	4	1	0	2	0
CFDA	—	—	2	4	0	0	2	0	0
CFMB	—	—	—	3	0	1	1	0	0
CFV	—	—	—	—	2	1	2	1	0
CW	—	—	—	—	—	0	2	1	0
DOS	—	—	—	—	—	—	0	3	0
DTS	—	—	—	—	—	—	—	1	0
PKUP	—	—	—	—	—	—	—	—	0

shows the results of checking the ten features for conflicts. The features are considered in pairs, and each property of one of the features in a pair is checked against every property of the other feature. The checks are carried out using a database of about 45 system axioms expressed as constraints like those above. (In fact, the above constraints are special cases of constraints in the database involving all possible resolutions of a call.) In the table, the numbers indicate the number of pairs of properties that resulted in a conflict when checking the pair of features against each other. Some entries are blank to avoid duplication. In some cases when more than one conflict is reported for a pair of features, the conflicts are for similar reasons but involve different pairs of properties. For example, the property `CFBL_Normal_Operation_1` mentioned above states the conditions under which a call must be forwarded. This property conflicts with two CFV properties, one that prohibits a call from being forwarded when CFV is deactivated by the subscriber, and one that prohibits a call from being forwarded when the call has already been forwarded before (e.g., because of a forwarding loop).

The tool has a variety of options; the results reported on here were done using the default settings. In the default case, for any pair of properties, the x occurring in the first property is considered to be the same as the x in the second property, and similarly for y and z. The system axioms are, however, instantiated in all possible ways. An average size check, for example checking ACR against CFBL which includes 18 pairwise checks, takes 20 minutes on a SGI Challenge machine.

Under the default settings, the tool will first check that the two input properties can be enabled together. If not, there is no conflict. Otherwise the conflict check is completed. Options provided in the tool include enhancements for greater efficiency and for more complete coverage in finding conflicts. One option for more comprehensive checks is the capability to provide alternative *variable bindings*. For example, x in a property of one feature can be bound to y in another.

It is possible to increase the effectiveness of the conflict checks by adding new predicates and new arguments to existing predicates so that properties can be expressed more precisely. For example, we write $busy(x)$ for x hearing a busy signal, but writing $busy(x, y)$ to mean that x hears a busy signal in response to an attempt to call y would be more precise. There is, however, a tradeoff: making the set of predicates more complicated increases the execution time required for model checking. We have attempted to keep the set of predicates simple and increase the precision carefully as needed.

6 Related Work and Conclusions

A variety of approaches to solving the feature interaction problem start by specifying a basic implementation in the form of an automaton or finite state machine, or even a procedural description that can be easily translated to a finite state machine representation. Various kinds of analyses are performed on these representations to detect interactions and check for other properties of features.

In several approaches that use finite state machines or other procedural specifications, feature requirements are expressed as properties in a temporal logic. Model checking or other state exploration techniques are used to check that these properties hold of the specification. Interactions are detected when certain properties are not satisfied, or when "bad" states are found to be reachable. Examples of this approach include [2, 6, 7, 8, 11, 14, 15]. For a more complete survey of this and related approaches, see [12].

Temporal logic is sometimes used to specify transitions of a state machine directly [2, 9]. In this approach, the same logic is used for both specifying the system and expressing properties of it. Maintainability of this kind of description is likely to be easier than for more explicit state transition representations; however, the logics used in this work are limited to next-state descriptions, so no liveness properties can be expressed.

Another approach (cf. [1, 13]) to detecting interactions between features A and B, which are specified as state machines, is to form the composed systems $A//Switch$ and $A//B//Switch$, and check if the behavior (i.e., the sequences of events) of A differs in the two systems; if this is so, the behavior of A has been affected by the presence of B.

In our work, we have described a method for detecting feature conflicts where features are specified as a collection of temporal logic formulae or ω-automata, and interactions are discovered by finding pairs of specification formulae that are contradictory with respect to axioms about system behavior. We show how existing model checkers can be used to perform this test. As discussed earlier, the advantages of this approach are that it simplifies the maintenance of specifications and avoids any commitment to a particular implementation, allowing the detection of conflicts that have wider applicability. We have implemented this method, and applied it to the analysis of formal specifications derived from the Telcordia standards. Our experience so far has been that this detection process is reasonably efficient and quite accurate; for the set of features to which we have applied this method, we have been able to detect most of the interactions given in the Telcordia standards, as well as some new ones.

An important component of future work will be to handle more features. Note that adding feature specifications does not increase the complexity of each conflict check, but does multiply the number of pairwise checks that must be carried out if we want to check each new feature against all existing features. In order to address the problem of scaling up, we will address the tradeoff of efficiency vs. power in FIX. By power, we mean not only allowing a greater number of conflict checks, but also achieving more accuracy in detecting conflicts. Along these lines, we plan to investigate the extensions discussed in Section 3: alternative variable bindings and building more precision into the feature specifications themselves. In addition, we plan to incorporate checks that include more than two features at a time.

Acknowledgements: We would like to thank Margaret Smith for translating the Telcordia standards into the initial set of feature specifications. Margaret Smith, Gerard Holzmann, Mihalis Yannakakis, Carlos Puchol and Bob Kurshan provided several valuable suggestions and encouragement.

References

[1] A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne. SCF3$^{\text{TM}}$/Sculptor with Chisel: Requirements engineering for communications services. In K. Kimbler and L. G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 45–63. IOS Press, 1998.

[2] J. Blom, R. Bol, and L. Kempe. Automatic detection of feature interactions in temporal logic. In K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III*, pages 1–19. IOS Press, 1995.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Conference on Computer Aided Verification.* Springer-Verlag, 1996.

[4] J. R. Buchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology and Philosophy of Science.* Stanford Univ. Press, 1962.

[5] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS.* Springer-Verlag, 1981.

[6] P. Combes and S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 120–135. IOS Press, 1994.

[7] L. du Bousquet. Feature interaction detection using testing and model-checking, experience report. In *World Congress on Formal Methods.* Springer Verlag, 1999.

[8] M. Faci and L. Logrippo. Specifying features and analysing their interactions in a LOTOS environment. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 136–151. IOS Press, 1994.

[9] A. Gammelgaard and J.E. Kristensen. Interaction detection, a logical approach. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 178–196. IOS Press, 1994.

[10] R. H. Hardin, Z. Har'el, and R. P. Kurshan. COSPAN. In *Conference on Computer Aided Verification.* Springer-Verlag, 1996.

[11] J. Kamoun and L. Logrippo. Goal-oriented feature interaction detection in the intelligent network model. In K. Kimbler and L. G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 172–186. IOS Press, 1998.

[12] 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, October 1998.

[13] T.F. LaPorta, D. Lee, Y-J. Lin, and M. Yannakakis. Protocol feature interactions. In *Formal Description Techniques (FORTE-PSTV)*, 1998.

[14] F.J. Lin and Y-J. Lin. A building block approach to detecting and resolving feature interactions. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 86–119. IOS Press, 1994.

[15] M. Plath and M. Ryan. Plug-and-play features. In K. Kimbler and L. G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 150–164. IOS Press, 1998.

[16] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[17] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*. Springer-Verlag, 1982.

[18] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook on Theoretical Computer Science*, volume B. Elsevier Science, 1990.

[19] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Symposium on Logic in Computer Science*, pages 332–344, 1986.