Feature Specification and Automated Conflict Detection

AMY P. FELTY
University of Ottawa
and
KEDAR S. NAMJOSHI
Bell Laboratories

Large software systems, especially in the telecommunications field, are often specified as a collection of features. We present a formal specification language for describing features, and a method of automatically detecting conflicts ("undesirable interactions") amongst features at the *specification* stage. Conflict detection at this early stage can help prevent costly and time consuming problem fixes during implementation. Features are specified using 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. In addition, the model checking tools can 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), which uses the model checker COSPAN for the inconsistency check. We describe our experience in applying this tool to a collection of telecommunications feature specifications obtained from the Telcordia (Bellcore) standards. Using FIX, we were able to detect most known interactions and some new ones, fully automatically, in a few hours processing time.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies, languages*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques, mechanical verification*

General Terms: Design, Reliability, Verification

Additional Key Words and Phrases: Feature interaction, telecommunications software and systems, linear temporal logic

A preliminary version of this article appeared in *Feature Interactions in Telecommunications and Software Systems VI*, M. Calder and E. Magill, Eds. IOS Press, 2000, pp. 179–192.
Authors' addresses: A. P. Felty, School of Information Technology and Engineering, University of Ottawa, 800 King Edward Ave., Ottawa, Ontario K1N 6N5, Canada; email: afelty@site.uottawa.ca; K. S. Namjoshi, Bell Laboratories, Lucent Technologies, 600-700 Mountain Avenue, Murray Hill, NJ 07974; email: kedar@research.bell-labs.com.

1. INTRODUCTION

Telecommunications services are marketed to customers by grouping together *features* such as call-waiting and call-forwarding. As the grouping is flexible, an individual feature is specified without knowledge of which other features it may be grouped with [Tel 1996]. This facilitates modular design and implementation; however, problems arise when concurrently active features in a group attempt to satisfy conflicting requirements. Implementors may resolve such conflicts as they are encountered in different ways, leading to unpredictable behavior in the system as a whole. Moreover, it is costly and time consuming to detect and fix such conflicts during or after implementation. It is therefore essential to detect and resolve such feature conflicts as early as possible, preferably at the specification stage itself.

With this motivation, we have developed a formal feature specification language, and a method of automatically detecting feature conflicts at the specification stage. We have implemented this method in a detection tool called FIX (for Feature Interaction eXtractor). 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 formulas has the nice property that it abstracts from specific state-machine implementations, allowing any implementation that satisfies the specifications.

The natural way to define a feature conflict is that the feature specifications represent mutually inconsistent properties; that is, no program exists that can implement both features. This is a question about whether the conjunction of two feature specifications is *realizable*. As discussed in Section 4.1, we also need to include *axioms* about the underlying system. The system axioms describe properties that should be true of *any* reasonable system implementation. 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 specified in the same specification language as the features. Specifying the system by axioms has the same nice property that it abstracts from particular implementations, resulting in conflict reports that have wider applicability.

Realizability checking for linear-time temporal properties differs from satisfiability checking, since it distinguishes between program and environment actions. It is also a hard problem, which is 2EXPTIME-complete [Pnueli and Rosner 1989]. Its solution requires the transformation of a linear time property to a branching time formula, which is then checked for satisfiability. This method is currently infeasible in practice, due to the lack of tools that are capable of handling large formulas efficiently. We take the approach, therefore, of approximating realizability by a constrained satisfiability problem. By considering systems with a fixed number of entities (i.e., telephones), feature specifications become propositional formulas, and this constrained satisfiability check can be performed automatically and efficiently with model checking tools. Our tool, FIX, reads in the specifica-

tions, converts them to ω-automata, and uses the model checking tool COSPAN [Hardin et al. 1996] to perform the satisfiability test. This detection process is fully automated. FIX provides *witness* computations for either outcome. If no conflict is detected, the witness describes a computation where both feature specifications hold; examining this computation often reveals gaps in the assumptions about the system that need to be filled in by modifying, or adding to, the system axioms. If a conflict is detected, a scenario is generated which describes a computation where the features conflict. By examining this scenario, one can determine either the proper resolution of the conflict, or whether this is a spurious conflict created by specifications that are too strong, and which need to be modified. Our specification method makes it easy to specify dynamic (i.e., state dependent) priorities between conflicting features, which are used to resolve conflicts.

Our experience so far has been that this detection process is reasonably efficient and quite accurate. The process of debugging the system axioms and the feature specifications, as described above, converges rapidly. We have applied this method to a large set of feature specifications from the Telcordia (Bellcore) standards, which were developed as part of a significant model checking project [Holzmann and Smith 2000]. For these features, we have been able to detect, in a matter of hours, most of the interactions given in the Telcordia (Bellcore) standards, as well as new ones.

A telecommunications system is, in a sense, an extreme example of designing with features. Our method has proved to be quite successful for these systems. It should be noted, though, that neither the specification language, nor the detection method, are specialized to telecommunications systems. Many other software systems are specified at an early stage of design as a collection of features. For instance, a user interface may be specified as a set of requirements of the form: "for this sequence of actions, the following response must occur," which fits our general scheme. We believe, therefore, that our techniques for the early detection of conflicts can be applied to a wide range of systems.

The rest of the article is structured as follows. Section 2 contains a short background on temporal logic, ω-automata and model checking. We motivate and define our specification language in Section 3. The precise formulation of feature conflict and the detection method is described in Section 4. The FIX tool is described in Section 5. The application of FIX to the Telcordia feature specifications is discussed in Section 6. The article concludes with a discussion of related work in Section 7.

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 Pnueli [1977]. Formulas 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 our current application domain of telephony networks. Formally, LTL formulas 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 formulas f and g, $(f \wedge g)$ (read as "f and g") and $\neg(f)$ (read as "not f") are formulas,
(3) For formulas f and g, $\mathsf{X}(f)$ (read as "next-time f") and $(f \ \mathsf{U} \ g)$ (read as "f until g") are formulas.

The temporal operators are X and U. Formulas are interpreted over infinite sequences of atomic proposition valuations. Such a sequence is defined as a function from $\mathbf{N}$ to 2^{AP} – for a sequence σ, $\sigma(i)$ is the subset of propositions that are true at position i. We write $\sigma, i \models f$ to mean that the sequence σ *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 is 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,
(3) $\sigma, i \models (f \wedge g)$ iff both $\sigma, i \models f$ and $\sigma, i \models g$ are true,
(4) $\sigma, i \models \mathsf{X}(f)$ iff $\sigma, i+1 \models f$,
(5) $\sigma, i \models (f \ \mathsf{U} \ g)$ 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 operators can be defined in terms of these base operators: $(f \vee g)$ is $\neg(\neg f \wedge \neg g)$; $(f \Rightarrow g)$ is $\neg f \vee g$; $\mathsf{F}(g)$ ("eventually g") is $(\mathit{true} \ \mathsf{U} \ g)$; $\mathsf{G}(f)$ ("always f") is $\neg\mathsf{F}(\neg f)$, and $(f \ \mathsf{W} \ g)$ ("f holds unless g") is $(\mathsf{G}(f) \vee (f \ \mathsf{U} \ g))$.

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 [Buchi 1962] 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*,
- Δ, a subset of $\subseteq S \times \Sigma \times S$, is the *transition relation*,
- I, a nonempty subset of S, is the set of *initial* states,
- F, a subset of 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 translation from LTL formulas to equivalent Büchi automata that is exponential in the worst case; see Thomas [1990] for a survey.

2.3 Model Checking

A program generates a set of computation sequences. For reactive programs where *nontermination* is desirable, such as operating systems and telephony protocols,

the sequences are infinite, in general; hence, temporal logic or Büchi automata may be used to describe program properties. 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 [Clarke and Emerson 1981; Queille and Sifakis 1982] can be used to determine whether 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 $\mathcal{L}(\mathit{Program}) \subseteq \mathcal{L}(\mathit{Property})$ [Vardi and Wolper 1986]. This question is typically decided by forming an automaton *NProperty* for the negation of the property, and algorithmically checking whether the product automaton $\mathit{Program} \times \mathit{NProperty}$ has an empty language.

Model Checking tools based on language containment include COSPAN [Hardin et al. 1996] and VIS [Brayton et al. 1996]. If the specification fails to hold of the program, the tool generates a computation that is a *witness* to this failure; that is, a computation in $\mathcal{L}(\mathit{Program})$ that is not in $\mathcal{L}(\mathit{Property})$. 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.

In specifying features, we began with the informal description, mostly in the form of English text found in the Telcordia (Bellcore) standards [Tel 1996]. Of course, the process of going from informal to formal specifications itself cannot be formalized, so care must be taken to correctly express the contents of the informal description. This section describes our formal specification language. Section 5 includes examples which illustrate how this language is used by providing formal specifications along with the informal descriptions that they were derived from; it also describes how the FIX tool can be used to help debug feature specifications to increase their accuracy.

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 and temporal operators (e.g., "and", "whenever", "even-

tually"). This is a pattern that is repeated throughout the Telcordia specification set. Hence, we believe that a particularly appropriate way of specifying a feature is by a collection of temporal formulas (or automata), defined over a set of predicates that denote states or events of the system.

Our specification notation is a sugared version of LTL. Each feature is specified separately, as a collection of temporal properties. The properties are defined in terms of predicates that indicate relationships between entities in the system. The feature specification also contains definitions for basic and derived predicates that are used in the properties. Concretely, each feature is placed in a separate file; for instance, call forwarding is specified in the file "call_forwarding.spec". We use the symbols `+`,`&`,`~`,`=>` to textually denote the Boolean operators $\vee, \wedge, \neg, \Rightarrow$ respectively.

There are two predefined predicates: $eq(x, y)$, which denotes equality of the entities x and y and, for each feature F, a predicate $disable_F(x)$, 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.

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

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. Variables such as x, y appearing in the predicates of the property specification have scope that is local to the property, and are implicitly universally quantified; that is, the temporal property should be true for every value of x, y in a particular system. 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 pattern holds, it is followed by the postcondition pattern".

The precondition has the following informal reading: "$e0$ holds, followed by a period where $(p0 \wedge \neg e1)$ is true, then $e1$ holds, followed by a period where $(p1 \wedge \neg e2)$ is true, etc., until eN holds." In extended regular expression notation, this can be written succinctly as $e0; (p0 \wedge \neg e1)^*; e1; (p1 \wedge \neg e2)^*; \ldots; eN$. 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. An empty precondition part defaults to the

precondition *true*.

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 $(p \mathsf{U} (r \vee d))$; with `unless` in place of `until`, it corresponds to the LTL formula $(p \mathsf{W} (r \vee d))$. Although 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 associated with the general form is to consider its negation: the property is false of an infinite sequence iff there is a point where the precondition pattern holds, but is not followed by the postcondition pattern. To illustrate the translation, consider the property below.

```
property Simple
{
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 formulas 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.

Our specification format was chosen, in part, because it is easy to translate a property specification to an automaton. We show first how to translate a property to an automaton that accepts its *negation*. The translated automaton has size linear in the size of the property, so that model checking (see Section 2.3) can be done efficiently – in time linear in the program size, and linear in the property size. Thus, the same properties that are used for early conflict detection can be used to efficiently model check actual implementations.

The negation of the simple property above is expressed by the nondeterministic Büchi automaton shown in Figure 1. In the figure, states are represented by circles, the transition relation is defined by the conditions on the arrows between circles, and accepting states are represented by concentric circles. The state labeled with $S0$ is the initial state. The automaton at state $S0$ chooses (nondeterministically) some point on a computation, checks that the precondition holds from that point (states $S1, S2$), and that the postcondition fails thereafter (i.e., the automaton gets stuck in states $S2$ or $S3$). The accepting states of the automaton are $\{S2, S3\}$ – the automaton stays in $S2$ if neither the response r or the discharge d hold and the

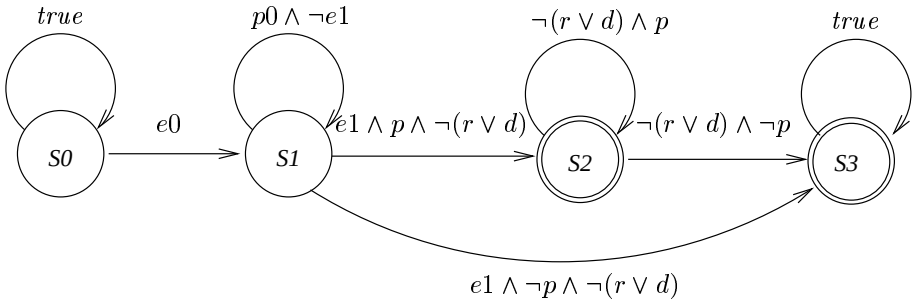

Fig. 1. Automaton for the negation of the simple property.

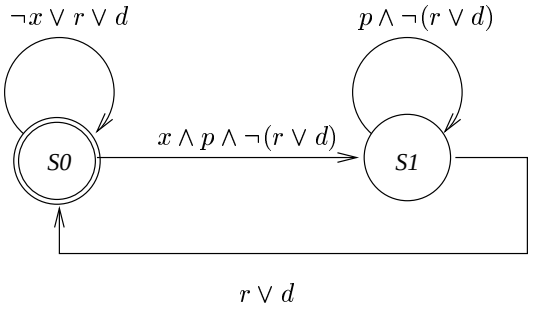

Fig. 2. Automaton for the positive version of the simple property.

persistence condition p holds, and moves to $S3$ and stays there if the persistence condition fails before the response or discharge can hold. The automaton for the general case has the same linear form, with $n+2$ states for a precondition with n events. If "until" is replaced with "unless" in the postcondition, the accepting set becomes just $\{S3\}$, since the unless property is satisfied if the automaton stays in $S2$ forever.

For our purposes, we also need the automaton for the property itself. This could be obtained by negating the automaton given above, but algorithms for negating Büchi automata are quite complex (cf. Thomas [1990]), so we prefer a direct construction. First, we construct a deterministic automaton A_{pre} that recognizes *all* the points on a computation where the precondition holds. Then, we form the automaton for $\mathsf{G}(A_{pre}.accept \Rightarrow (p \mathsf{U} (r \vee d)))$. This automaton is shown in Figure 2, where x stands for $A_{pre}.accept$. This automaton is run in parallel with A_{pre} (by forming the product automaton) to get the automaton for the property.

To construct the automaton A_{pre}, we take the nondeterministic automaton on finite strings that is induced by states $S0, S1$ in Figure 1, with the acceptance condition $(state = S1) \wedge e1$. This nondeterministic automaton has an accepting run to every point on a computation where the precondition holds. Now, we apply the subset construction to this automaton to determinize it and form A_{pre}. The deterministic automaton A_{pre}, by construction, has a single run on a computation that signals acceptance at every point where the precondition holds.

We have shown how features may be represented by formulas 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`.

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 formulas. We say that a feature is enabled if one of the properties of the feature is enabled. We identify the name of a feature, say A, with its specification in terms of properties.

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 the specifications for both A and B.

Thus, feature conflict is essentially a *realizability* question: features A and B conflict if and only if a program realizing their joint specification $A \wedge B$ does not exist. Notice that we are interested here in just the decision question: does such a program exist? The problem of synthesizing such a program is a classical problem which has potential practical applications. Any program that satisfies $A \wedge B$ will be an open reactive program [Harel and Pnueli 1985] which constantly interacts with its environment. For instance, a program satisfying the call-forwarding specification will have to respond to incoming call events and produce outgoing call events. Unfortunately, the realizability question for open reactive programs has a very high complexity (2EXPTIME-hard), and the known solutions are based on showing satisfiability of a branching time formula obtained from the linear time formulas describing A and B [Pnueli and Rosner 1989]. Instead of using these decision algorithms, for efficiency reasons, we opt to approximate branching time satisfiability by constrained linear time satisfiability. Linear time satisfiability checking is supported by many model checking tools, such as the tool we used, COSPAN. In later sections, we describe in more detail how COSPAN is used to perform these checks. The availability of such tools for LTL, but not for other temporal logics, such as branching time logics like CTL [Clarke and Emerson 1981] and CTL* [Emerson and Halpern 1986], or the Temporal Logic of Actions (TLA) [Lamport 1994], was a major factor in our choice of LTL as the specification language.

t turns out that checking $A \wedge B$ for linear time satisfiability is too much of an approximation to the branching time formulation of realizability. We need to rule out several paths in the branching tree that cause $A \wedge B$ to be satisfied

trivially. Our final criterion is described below as the end result of refining a series of approximations, starting with linear time satisfiability. This criterion suffices to detect a number of feature conflict problems, as described later in Section 5. The initial formulation is in terms of linear time satisfiability.

Definition 4.1. Features A and B conflict iff the formula $(A \wedge B)$ is unsatisfiable; that is, 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 below.

$A : \mathsf{G}(\mathit{calls}(a,b) \;\Rightarrow\; \mathsf{F}(\mathit{connected}(a,b) \;\vee\; \mathit{disconnect}(a)))$
("Whenever a calls b, a and b are connected, unless a disconnects"),

$B : \mathsf{G}(\mathit{calls}(a,b) \;\Rightarrow\; \mathsf{F}(\mathit{forwards}(a,b,c) \;\vee\; \mathit{disconnect}(a)))$
("Whenever a calls b, the call is forwarded to c, unless a disconnects").

Informally, these specifications are conflicting, since forwarding from b and connecting to b should not both happen for the same call. Yet the conjunction of the formulas is satisfiable: consider the computation in which $\mathit{calls}(a,b)$ is always false! The problem here is that it is always possible to trivially satisfy a feature specification if 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.

Definition 4.2. 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 nonconflicting! 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.

Definition 4.3. 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 nonconflicting! Consider the computation in which after $\mathit{calls}(a,b)$ holds, $\mathit{disconnect}(a)$ is true before either of the predicates $\mathit{connected}(a,b)$ or $\mathit{forwards}(a,b,c)$ holds. Both specifications are thus satisfied trivially because the discharge condition is asserted before any useful actions are performed. 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*, that is, enabled but not satisfied. The following definition is the one that we use in our detection method.

Definition 4.4 *(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) A discharge condition does not occur while the feature is pending,

some feature property does not hold.

Conditions 2 and 3 can be expressed with simple formulas 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. Let A and B be two features, let Ax denote the system axioms, and C_{AB} the constraints given by conditions 2 and 3 of Definition 4.4. The inconsistency check can be written as $\overline{\mathcal{L}(A)} \cap \overline{\mathcal{L}(B)} \cap \mathcal{L}(Ax) \cap \mathcal{L}(C_{AB}) = \emptyset$, which is equivalent to $\mathcal{L}(Ax) \cap \mathcal{L}(C_{AB}) \subseteq \mathcal{L}(A) \cup \mathcal{L}(B)$. This is the form used in our implementation.

The FIX tool that we have developed uses the model checker COSPAN [Hardin et al. 1996] for the conflict check. In COSPAN, both properties and constraints are represented by ω-automata. FIX translates the constraints Ax and the feature specifications A, B into COSPAN automata that accept the specified languages, as explained in Section 3. Each feature is translated to a parameterized automaton (parameterized by the variables appearing in the properties) 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 to obtain the automaton for C_{AB}.

The model checker declares failure if the set inclusion above is false; that is, if the properties *do not* conflict. The nonconflict 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: $\mathcal{L}(Ax) \cap \mathcal{L}(C_{AB}) \cap \mathcal{L}(A) \subseteq \mathcal{L}(B)$. As there is a conflict,

this check must fail,[1] so the model checker produces a computation that satisfies Ax, C_{AB} and A but does not satisfy B. This computation describes a scenario in which the system axioms hold, both features are enabled together infinitely often and A holds, but B does not hold.

5. FIX: A CONFLICT DETECTION TOOL

The FIX tool is used to both specify the desired properties of features (using the language described in Section 3) and to detect conflicts among them, as described in the previous section. FIX is intended to be used at the design and specification stage of the development of new features. Here, we describe the tool in more detail and illustrate its use with the aid of examples from well-known features.

The first step in using FIX is to provide a set of properties that specifies the desired behavior of the new feature. The specification language described in Section 3 was designed so that properties of features could be specified as naturally and directly as possible. The main components of the property templates — events, persisting conditions, required resolutions or conditions that must never occur after a set of preconditions are met, and exception or discharge conditions — were chosen because all of the properties of the well-known features that we examined contained some or all of them. In addition, they are concepts that are easily understood by a designer of a feature. There is no need to know the languages of linear temporal logic or Büchi automaton into which the properties will be translated. At the same time, there is a close enough correspondence to these formal languages that the properties are easily translated (as shown in Section 3). When developing a new specification, the user has the freedom to introduce new predicates as needed. The introduction of new predicates usually requires new system axioms to be added or existing axioms to be updated, which can also be done at this stage.

The conflict check is the central operation of FIX. As described in the previous section, there are two kinds of checks: the inconsistency check, and the check which produces a "conflict witness" once an inconsistency has been detected. For both kinds of checks, FIX expects two properties, A and B, as input. The system axioms Ax are fixed and the auxiliary automaton C_{AB} is created automatically from A and B. The second step in using FIX is to use the first kind of check as a debugging aid. In particular, each property of the new feature can be checked for direct conflict with the system axioms. To perform a check of a single property A against the system axioms, we simply instantiate B as the "always true" property, specified as:

```
property true_prop
{
----------------------
persists: true
}
```

A conflict arising from such a check represents a bug in either the property or in the system axioms. The user must then either modify A or Ax and repeat the check.

[1] This check will succeed only in the pathological situation that A always fails under the condition $\mathcal{L}(Ax) \cap \mathcal{L}(C_{AB})$. Then it is possible to produce a computation satisfying both Ax, C_{AB} but not A by checking $\mathcal{L}(Ax) \cap \mathcal{L}(C_{AB}) \subseteq \mathcal{L}(A)$.

Since the conflict checks are conclusive, once the initial specification phase is complete, the user can be sure that there are no conflicts between the feature properties and the system axioms. On the other hand, because a result of "no conflict" is inconclusive, there is no guarantee that all potential conflicts between features will be found. Of course it is important to make the specifications strong enough to detect as many conflicts as possible. FIX can provide support for this task in a second debugging phase. The user can choose one or two previously specified features, check each property of the new feature against each property of the existing features, and examine the witness computations that result when two properties do not conflict. In our experience, finding nonconflicting pairs that should really be conflicting can help the user find and strengthen specification formulas that were not originally stated as strongly as they could be. Also, this phase often reveals system axioms that are not strong enough, particularly those that were just added as a result of new predicates introduced by the new feature.

Once the user has gained enough assurance that the specification and system axioms are correct, properties of the new feature can be checked against *all* other features fully automatically. At this stage, only the conflicts are important and the conflict witnesses are useful for understanding and determining how to correct them.

To illustrate, we take some examples from our case study, which is described more fully in the next section. Two of the features that we consider are Call Forwarding Busy Line (CFBL) and Anonymous Call Rejection (ACR). For CFBL, the subscriber gives a number to which all calls will be forwarded when the subscriber's line is busy. Calls to a subscriber of the ACR feature will not go through when the caller prevents her number from being displayed on the subscriber's caller ID device. For this example, we assume that CFBL was previously defined and ACR is a new feature to be specified. The following is one of three properties of CFBL.

```
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 x subscribes to CFBL, x is not idle, all previously forwarded calls from x to z have terminated, x and z are on the same switch, the incoming call from y has been forwarded at most five times and 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. 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 at which a formula becomes true. For example, $call\_req(x, y)$ 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 forwarding.

Two of the system axioms present in the system after CFBL is defined, but before ACR is added are the following.

```
constraint call_req_not_resolution
{
---------------
persists: call_req(x,y) => (~busy_tone(y) & ~forwarding(x,y,_))
}

constraint distinct_resolutions
{
---------------
persists: ~(forwarding(_,y,_) & busy_tone(y))
}
```

The information expressed here is that (1) a call request is distinct from a call resolution and (2) that two call resolutions cannot occur at the same time. For this example, receiving a busy tone and having a call forwarded are the two resolutions considered so far. The first property states that at any point in time when x has an outstanding call request from y, y is neither receiving a busy tone nor having its call to x forwarded. The second property states that a call from y is not being forwarded at the same time that y is receiving a busy tone. It is possible for a call to have several steps to its resolution. For example, a call from y to x may be forwarded to z followed by y receiving a busy tone after it is determined that z is busy, but the forwarding and receiving of the busy tone do not happen at the same time.

```
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)
}
```

The property above is one of six properties we add to specify ACR. 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.

This property and the CFBL property stated above 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 the preconditions of both properties 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.

When we run the inconsistency check on these two properties using the system axioms that we have discussed so far, no conflict is detected. One possible

witness computation that may arise is one in which the call from y to x is forwarded and given the ACR denial announcement at the same time. This is possible because the specification of ACR has introduced three new predicates (ACR, $resources\_for\_ACR\_annc$, ACR_annc) that have not yet been incorporated into the system axioms. In order for FIX to detect this particular conflict, it is enough to integrate ACR_annc as a new kind of call resolution. One way to do this is to update the first constraint listed above, and replace the second with three new ones as follows.

```
constraint call_req_not_resolution
{
---------------
persists: call_req(x,y) => (~busy_tone(y) & ~forwarding(x,y,_) &
                            ~ACR_annc(y,x))
}

constraint resolution_forwarding_only
{
---------------
persists: forwarding(x,y,_) => (~busy_tone(y) & ~ACR_annc(y,x))
}

constraint resolution_busy_tone_only
{
---------------
persists: busy_tone(y) => (~forwarding(x,y,_) & ~ACR_annc(y,x))
}

constraint resolution_ACR_annc_only
{
---------------
persists: ACR_annc(y,x) => (~busy_tone(y) & ~forwarding(x,y,_))
}
```

The last three constraints show a fairly general form for distinguishing call resolutions; each time a new call resolution predicate is introduced, one new constraint must be introduced distinguishing it from all the rest, and the old constraints must be updated to include the new resolution in the persists condition.

Note that although we have to debug and maintain an axiom system, as was mentioned earlier, we do not have to maintain an implementation. We have described here how to "debug" the axioms, and in our experience, although finding bugs in axioms is different than debugging an implementation, it is no harder or easier. Correcting an error in an implementation requires considering the effect of the correction on the rest of the implementation and making sure it does not introduce new errors, while correcting errors in axioms affects only one axiom at a time. Of course, any single correction to an axiom has a global effect, but we believe it less error prone than changing an implementation.

We do not try to produce a "complete" set of axioms in any sense: we introduce just enough axioms to detect meaningful conflicts. Our case study described in the next section shows that we are able to do so. Since both system axioms and feature

specification formulas are expressed in the same formalism, debugging and maintaining these two kinds of formulas is the same activity. The distinction between the two is only informal; system axioms express properties that should hold no matter what features are introduced, while specification formulas formally express requirements of a particular feature.

This conflict between ACR and CFBL is a known conflict whose resolution is described in the Telcordia documents. When the presentation of the number is not allowed, ACR should take precedence over CFBL and the denial announcement should be given. In our setting, we can express this kind of precedence by adding the condition $(ACR(x) \Rightarrow DN\_allowed(y))$ as an additional conjunct to the event part of the CFBL property. This conjunct will be false exactly when both $ACR(x)$ and $\sim DN\_allowed(y)$ hold, thus falsifying the entire precondition of the CFBL property in exactly the cases when the ACR property should take precedence. When solving interactions between two features is simply a case of establishing a priority between them, it is actually not necessary to modify the specifications. As mentioned, FIX provides a mechanism for specifying priorities globally, which we illustrate later.

FIX has a variety of options. In the default case, for any pair of properties, the x occurring in both properties is instantiated by the same constant, and similarly for y and z. The system axioms are, however, instantiated in all possible ways using three constants.

Also as part of the default, FIX 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 instantiated with the same constant as 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 $busy\_tone(x)$ for x hearing a busy signal, but writing $busy\_tone(x, y)$ to mean that x hears a busy signal in response to an attempt to call y would be more precise. There is, however, a trade-off: 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. CASE STUDY

We have applied our tool to a collection of feature specifications derived from the Telcordia standards [Tel 1996]. We report on the results for ten of these features, each checked against the nine others.

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

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 50 system axioms expressed as constraints

Table I. 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.

like those discussed in the previous section. The constraints in the previous section are special cases of a group of constraints in the database that have a similar form, but involve all possible resolutions of a call. Handling the 10 features of our case study required 13 possible call resolutions: 2 kinds of announcements, 5 kinds of tones heard by the caller, 4 kinds of rings, forwarding, and successful connection.

The system axioms include roughly three other groups of constraints. The second group was adapted from an English description in the Telcordia documents specifying what it means for a particular entity to be busy or idle. The originator of a call is said to be busy from the time the phone goes off hook until it goes back on hook, whether or not the call is successfully completed. The party who is called is said to be busy from the time that ringing starts until either the call is aborted by the caller, or the call terminates normally. Approximately 15 constraints describe these concepts.

A third group of constraints, also taken directly from the Telcordia documents, specifies properties for the predicates introduced specifically for call waiting, which is one of the more complicated features. For this feature, the notions of stable calls and calls that are on hold are important. For example, if no call is in process for an

Table II. 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

entity, then that entity is neither in a stable call state nor an unstable call state. An answered call is a stable call and a partially dialed call is an unstable call. Other important constraints in this group state that the time spent on hold is distinct from the call request and call resolution phases of a call.

The final group of constraints deals with forwarded calls. When forwarding occurs, it is never the final resolution of the call and there are restrictions on what the remaining steps can be. The following is an example from this group stating that a forwarded call should never subsequently be denied by the ACR feature.

```
constraint forwarding_not_followed_by_ACR_annc
{
event: offhook(x) & forwarding(_,x,_)
--------------------------
persists: ~ACR_annc(x,_)
unless: onhook(x)
}
```

The majority of the system axioms can be expressed using the simple form illustrated in the last section where only the persists condition is important. The above constraint is an example showing that a more complicated sequence is sometimes needed to express a constraint.

Table II shows the results of checking the ten features for conflicts. 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. The results reported on here were done using the default settings of FIX. An average size check, for example checking ACR against CFBL which includes 18 pairwise checks, takes 20 minutes on a SGI Challenge machine.

We examine the pair of features ACR and CFBL in more detail to further illustrate the conflicts that FIX detects. CFBL is specified by three properties. In addition to the property specifying normal operation given in the previous section, the following two properties specify exceptions to normal operation.

```
property CFBL_Exception_1_to_Normal_Operation_1
{
event: CFBL(x) & ~idle(x) & forwarding(x,_,z) & same_switch(x,z) &
       call_req(x,y)
--------------------------
persists: call_req(x,y)
```

```
until: busy_tone(y) & ~forwarding(x,y,z)
discharge: onhook(y)
}

property CFBL_Exception_2_to_Normal_Operation_1
{
event: CFBL(x) & ~idle(x) & ~le_five_forwards(y) & call_req(x,y)
---------------------------
persists: call_req(x,y)
until: busy_tone(y) & ~forwarding(x,y,z)
discharge: onhook(y)
}
```

Like the property already given, these properties also consider the case when, initially, x is not idle and there is an incoming call from y. The first exception handles the case when another call to x is in the process of being forwarded at the time when y calls. The second exception handles the case when the incoming call from y has been forwarded more than five times. This condition is often caused by a forwarding loop. In both of these cases, the call should not be forwarded. Instead, y should receive a busy tone. Both of these properties conflict with `ACR_Normal_Operation_3`, and in both cases it is because the ACR property requires the caller to receive the ACR denial announcement, while CFBL requires the caller to receive a busy tone. As before, ACR should have precedence over CFBL in these cases, and the conflicts can be resolved by adding $(ACR(x) \Rightarrow DN\_allowed(y))$ to the event parts of these properties.

Next, consider the following property of ACR, which also specifies normal operation.

```
property ACR_Normal_Operation_2
{
event: ACR(x) & call_req(x,y) & DN_allowed(y)
-----------------------
persists: call_req(x,y)
until: audible_ringing(y) + busy_tone(y)
discharge: answer(x,y) + onhook(y)
}
```

The main difference with the other ACR normal operation property is that the presentation of y's number is allowed. In this case, the call should proceed and y should eventually receive either a ringing tone or a busy tone. Note that there is no conflict of this property with the exception cases for CFBL. When x is not idle and y's number can be presented, the call must be resolved by y receiving a busy tone.

Note, however, that the new ACR property does conflict with the property `CFBL_Normal_Operation_1` because, once again, the two features require that the incoming call be resolved in different ways: ACR requires the caller to receive either ringing or a busy tone, while CFBL requires that the call be forwarded. The Telcordia documents specify that when the presentation of the number is allowed, the call should be processed according to the requirements of CFBL. We can resolve this conflict by adding $\sim CFBL(x)$ to the event part of `ACR_Normal_Operation_2`.

This ACR property can be ignored when the CFBL feature is active.

All the properties we have stated so far are liveness properties, specifying sequences of events that must occur under certain preconditions. Two of the six properties specifying ACR are safety properties indicating sequences of events that must never occur.

```
property ACR_Normal_Operation_1
{
event: ACR(x) & call_req(x,y) & DN_allowed(y)
-----------------------
persists: ~ACR_annc(y,x)
discharge: onhook(y) + disconnect(y,x)
}

property ACR_Normal_Operation_4
{
event: ACR(x) & call_req(x,y) & ~DN_allowed(y)
-----------------------
persists: call_req(x,y) & ~busy_tone(y) & ~audible_ringing(y)
unless: ACR_annc(y,x)
discharge: onhook(y)
}
```

The first states that when y's number can be presented, there is no ACR denial announcement given during the duration of the call. The call ends either by y going back on hook or being disconnected by the system. Recall that the default when there is no unless/until keyword is `unless: false`. This property is fairly specific to ACR and there is no conflict with CFBL.

The second property expresses the requirement that there be no busy or ringing tone given to y when the presentation of y's number is restricted. This property conflicts with the two CFBL properties whose resolution is that a busy tone must be given. These conflicts are already resolved by the solution given earlier which adds the conjunct $(ACR(x) \Rightarrow DN\_allowed(y))$ to all the CFBL properties.

In these examples, selective disabling of the features to resolve interactions was done by adjusting the specifications. An alternative method is to use the predefined $disable_F(x)$ predicate, and insert the following constraint into the set of system constraints. The constraint ensures that the appropriate feature is disabled depending on the state of the system.

```
constraint resolve_ACR_CFBL_interactions
{
event: ACR(x) & CFBL(x) & call_req(x,y)
---------------------------
persists: (DN_allowed(y) => disable_ACR(x)) &
          (~DN_allowed(y) => disable_CFBL(x))
unless: ~call_req(x,y)
}
```

These examples have illustrated four of the six ACR properties and six of the eight conflicts between ACR and CFBL. The remaining two properties are similar to `ACR_Normal_Operation_2` and when checked against `CFBL_Normal_Operation_1`,

produce conflicts similar to those already discussed

7. RELATED WORK AND CONCLUSIONS

Several approaches have been proposed for the conflict detection problem. There are two main categories based on the specification formalism: state machine based methods and temporal logic based methods. Our approach falls into the temporal logic category. We describe the two approaches below, arguing that the temporal logic method has several advantages over the state machine approach.

In several specification methods [Blom et al. 1995; Combes and Pickin 1994; du Bousquet 1999; Faci and Logrippo 1994; Jonsson et al. 2000; Kamoun and Logrippo 1998; Khoumsi and Bevelo 2000; Lin and Lin 1994; Plath and Ryan 1998; Siddiqi and Atlee 2000], each feature is specified by a state machine. Interactions are detected by testing the composition of the machines, either for reachability of "bad" states, or for reachability of states where the features postulate conflicting actions on a new input, or against temporal properties specifying the feature behavior. For a more complete survey of this and related approaches, see Keck and Kuehn [1998].

In particular, Plath and Ryan [2001] describe a system based on extensions to the model checker SMV [McMillan 1993]. Features are built by layering changes on a base feature. The base feature is specified as a state machine, described implicitly in SMV syntax by a set of state variables and conditional assignments to those variables. The valuations of the state variables define the states of the machine, and the conditional assignments define the transitions. Each layer may introduce new state variables, with their own assignment statements, and additionally define a set of changes to the updates of existing variables. Feature interactions are detected by checking temporal properties of features against the composition of the state machines describing the features.

This approach uses existing model checking tools in a direct way, but it has two main disadvantages. In practical terms, there is repetition of work in specifying a feature both in temporal logic and as a state machine. It is necessary to check the consistency of the two specifications by model checking. Secondly, a state machine defines one particular implementation of the feature. Thus, if the features are reported as conflicting, it is unclear whether this conflict is specific to the particular state machine, or it exists in all implementations. Our approach addresses both of these difficulties. The first is eliminated by considering the temporal properties as being the only specification of a feature. The second one is avoided by the detection method. A conflict found by our method is applicable to *all* implementations that satisfy the system axioms and the individual feature specifications. Technically, this generalization means that, as discussed in Section 4.1, we are solving—albeit approximately—an instance of the more difficult realizability question.

The existing temporal logic approaches [Blom et al. 1995; Gammelgaard and Kristensen 1994] use only a subset of temporal logic, and their descriptions of features are essentially state machines presented in logical notation, so it is impossible to express liveness properties, for instance. A different approach (cf. Aho et al. [1998] and LaPorta et al. [1998]) to detecting interactions between features A, B specified as state machines, is to form the composed systems $A//Switch$ and $A//B//Switch$, and check if the behavior of A differs in the two systems: if this is so, the behavior

of A has been affected by the presence of B. While this is a promising method, it requires abstract models of the switch and of the features. Such models can be more difficult to create and maintain than logical specifications. Although we use automata in the conflict check, which can be viewed as state machines with fairness constraints, the translation from LTL formulas to automata occurs in a manner invisible to the end user. Furthermore, the behavior of a feature specified as a collection of temporal properties can often be restricted by adding (in effect, conjoining) another formula. It is much harder to get the same effect for a state machine description; the machine may have to be modified significantly in order to restrict its behavior. On the other hand, if it is indeed possible to describe the feature easily using a state machine, the machine can be encoded quite simply with temporal logic.

In our work, we have described a method for detecting feature conflicts where features are specified as a collection of temporal logic formulas or ω-automata, and interactions are discovered by finding pairs of specification formulas that are contradictory with respect to axioms about system behavior. We showed how existing model checkers can be used to perform this test. The main advantages of this approach are that (i) the specification language simplifies the maintenance of specifications, (ii) the method avoids any commitment to a particular implementation, which means that a detected conflict applies to all implementations, and (iii) it can be implemented to perform fully automated conflict detection, using existing model checkers in an effective manner. 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. For this set of features, our tool FIX is able to detect these interactions in a matter of a few hours of processing time.

An important component of future work is to handle more features, as well as to improve the performance of the tool. Adding feature specifications does not increase the complexity of each conflict check, which is still carried out pairwise among individual properties, but it does multiply the number of such checks that must be carried out if we want to check each new feature against all existing features. On the other hand, since the pairwise interaction checks can be run independently, it is feasible to use machines in a network in parallel to dramatically reduce the time needed to detect interactions. In order to address the problem of scaling up, we will address the trade-off 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 5: alternative variable bindings and building more precision into the feature specifications themselves. We also plan to incorporate checks that include more than two features at a time. Another line of research is suggested by the formulation of feature interaction as a realizability question (Section 4.1). It would be interesting to implement the full branching time solution and compare the results with those we have obtained using a linear time approximation. If the synthesis problem turns out to be efficiently solvable in practice, it would also be interesting

to investigate whether efficient programs meeting the feature specifications can be synthesized out of the requirements.

ACKNOWLEDGMENTS

We would like to thank Margaret Smith for much help in developing the formal feature specifications from the Telcordia standards. Margaret Smith, Gerard Holzmann, Mihalis Yannakakis, Carlos Puchol and Bob Kurshan provided several valuable suggestions and encouragement.

REFERENCES

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

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

Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A. L., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S. A., Khatri, S. P., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy, G., and Villa, T. 1996. VIS: A system for verification and synthesis. In *Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New York.

Buchi, J. R. 1962. On a decision method in restricted second-order arithmetic. In *1960 International Congress for Logic, Methodology and Philosophy of Science*. Stanford Univ. Press, Stanford, Calif.

Clarke, E. M., and Emerson, E. A. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*. Lecture Notes in Computer Science, vol. 131. Springer-Verlag, New York.

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

du Bousquet, L. 1999. Feature interaction detection using testing and model-checking, experience report. In *World Congress on Formal Methods*. Lecture Notes in Computer Science, vol. 1708. Springer Verlag.

Emerson, E. A., and Halpern, J. Y. 1986. "Sometimes" and "Not Never" revisited: on Branching versus Linear Time Temporal Logic. *J.ACM 33,* 1 (Jan.), 151–178.

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

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

Hardin, R. H., Har'el, Z., and Kurshan, R. P. 1996. COSPAN. In *Eighth Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New York.

Harel, D., and Pnueli, A. 1985. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, K. Apt, Ed. NATO Advanced Summer Institutes, vol. F-13. Springer-Verlag, New York, 477–498.

Holzmann, G. J., and Smith, M. H. 2000. Automating software feature interaction. *Bell Labs Tech. J. 5.*

Jonsson, B., Margaria, T., Naeser, G., Nyström, J., and Steffen, B. 2000. Incremental requirement specification for evolving systems. In *Feature Interactions in Telecommunications and Software Systems VI*, M. Calder and E. Magill, Eds. IOS Press, 145–162.

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

Keck, D. O., and Kuehn, P. J. 1998. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Trans. Softw. Eng. 24,* 10 (Oct.), 779–796.

Khoumsi, A., and Bevelo, R. J. 2000. A detection method developed after a thorough study of the contest held in 1998. In *Feature Interactions in Telecommunications and Software Systems VI*, M. Calder and E. Magill, Eds. IOS Press, 226–240.

Lamport, L. 1994. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst. 16,* 3 (May), 872–923.

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

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

McMillan, K. L. 1993. *Symbolic Model Checking.* Kluwer Academic Publishers.

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

Plath, M., and Ryan, M. 2001. Feature integration using a feature construct. *Sci. Comput. Prog. 41,* 1 (Sept.), 53–84.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif., 46–57.

Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York.

Queille, J. P., and Sifakis, J. 1982. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming.* Lecture Notes in Computer Science, vol. 137. Springer-Verlag, New York.

Siddiqi, S., and Atlee, J. M. 2000. A hybrid model for specifying features and detecting interactions. *Comput. Netw. 32*, 471–485.

TEL 1996. LATA switching systems generic requirements (LSSGR) FR-NWT-000064, 1992 edition. Feature requirements, including: SPCS capabilities and features, SR-504. Issue 1, May 1996, Telcordia/Bellcore.

Thomas, W. 1990. Automata on infinite objects. In *Handbook on Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. Elsevier Science, Amsterdam, The Netherlands.

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

Received February 2001; revised March 2003; accepted March 2003