Software & System Modeling (2005)
DOI 10.1007/s10270-005-0101-6

REGULAR PAPER

Nicolas Gorse · Luigi Logrippo · Jacques Sincennes

Formal detection of feature interactions with logic programming and LOTOS

Received: 5 February 2003 / Revised version: 4 September 2004 / Published online: 16 November 2005

Abstract This article addresses the problem of detecting feature interactions in the area of telephony systems design. The proposed approach consists of two phases: filtering and testing. The filtering phase detects possible interactions by identifying incoherencies in a logic specification of the main elements of the features, consisting of preconditions, triggers, results and constraints. If incoherencies are identified, then an interaction is suspected, test cases corresponding to the suspected interaction are generated and testing is applied to see if the interaction actually exists. Two case studies, carried out on established benchmarks, show that this approach gives good results in practice.

Keywords Telephony software · Feature interaction · Detection method · Formal techniques

1 Introduction

Today's telephony systems are usually defined as the composition of a base system and features [1].

By feature, we refer to the increment of functionality offered by a telephony system. A feature has the purpose of fulfilling certain user intentions in the context of a call. A telephony system is seen as a base implementation around which features are *orbiting*. Let us call $\mathcal{B}$ the base of the system and $\mathcal{F}_x$ a feature where x identifies the name of the feature, and the composition operator $\oplus$. The composition of a feature $\mathcal{F}_x$ with the base system $\mathcal{B}$ is then denoted: $\mathcal{B} \oplus \mathcal{F}_x$. Hence, a system decomposed into a base $\mathcal{B}$ and n features $\mathcal{F}_x$ is denoted: $\mathcal{B} \oplus \mathcal{F}_1 \oplus \mathcal{F}_2 \oplus \ldots \oplus \mathcal{F}_n$.

The advantage of such a view is that it separates the prime behavior from optional ones. This distinction facilitates the addition of new capabilities by eliminating the need to redesign or re-implement the whole system. However, the combination of features can lead to the so-called feature interaction problem. Some interactions are desirable and planned, others may have disruptive effects, in the sense that the user may not obtain the expected system behavior.

The reader of this paper is supposed to have some knowledge of the feature interaction problem. Abundant literature and a series of conferences have been dedicated [2–4] to this problem. In addition, a recent tutorial which contains abundant references is [5].

Feature interactions, already a problem in conventional telephony, are expected to become a serious obstacle for the deployment of the many features that will be possible in Internet telephony [4, 6, 7]. All interactions, wanted or not, should be known by the designer, who determines how they are dealt with, e.g. which features have the priority, which features can modify the behavior of others, which features need to be modified in order to accommodate others, etc. Analysis of interactions should be performed as early as possible in the development cycle, i.e. at the requirements level. This allows designers to be aware of problems before features are implemented, thus reducing development costs.

Because features can be arbitrarily complex, the general problem of feature interaction detection is computationally unsolvable and heuristic methods, which are incomplete, must be used. Many methods have been proposed for this purpose (see references given above). Benchmarks of known interactions can be used to assess the completeness of a given method [8].

Industrial methods to deal with feature interactions involve first of all the designer's experience and skill. They also involve very extensive testing of prototype systems. The aim of this research is to make this process more efficient,

Communicated by Robert France

N. Gorse (✉)
Université de Montréal, Dépt. d'informatique et recherche opérationnelle, Montréal, Canada
E-mail: gorsen@iro.umontreal.ca

L. Logrippo
Université du Québec en Outaouais, Dépt. d'informatique et ingénierie, Gatineau, Canada
E-mail: luigi@uqo.ca

L. Logrippo · J. Sincennes
University of Ottawa, School of information technology and engineering, Ottawa, Canada
E-mail: jack@site.uottawa.ca

by identifying the essential aspects of feature interaction. It should be noted that the industrial process we have mentioned will become more problematic in the future. One of the characteristics of Internet telephony will be the capability for users to program their own features. Obviously, users will need interactive help to catch common pitfalls in this process, and our method promises such help [9].

Symbolic execution or exhaustive testing of execu-table formal specifications of the features, in order to see if an undesirable situation can be reached, are established methods, but can be very costly, because of the extremely large number of possibilities to be considered in most cases (state explosion problem).

Filtering [10] has been proposed to alleviate such a problem. Filtering is a static analysis of feature specifications in order to see whether an interaction is possible. However, filtering by itself may not be able to consider all elements of the features being examined, because of their often complex behavior. If filtering detects the possibility of an interaction, symbolic execution or testing may have to be performed to see whether the interaction can really occur. Unfortunately, and for the same reason, it is also true that filtering may not detect all interactions. However, being able to automatically detect even a portion of possible interactions is still better than having to perform manual analysis for all cases.

Our method starts with a filtering stage, based on the idea that feature interactions arise from inconsistencies among the definitions of the various features in a system. Some important inconsistencies are identified and formally defined in this paper under the name of *incoherencies*. We show that it is possible to detect interactions by identifying, at the requirement level, combinations of features presenting incoherencies.

An example of incoherence is two features being triggered by the same event and meant to act upon this event in different and inconsistent ways (see *Incoming Call Screening* and *Call Forward on Busy* in Sect. 3.1.1). Another example is a feature triggering another one where their results present a contradiction (see *Call Forward on Busy* and *Outgoing Call Screening* in Sect. 3.2.1).

We have a filtering stage, where formal filtering rules are applied to a formal representation of the features to identify incoherencies between features. This stage is described in Sect. 3 and 4. If the filtering stage identifies an incoherence, a scenario-based validation can be performed using automatically derived test suites. For each incoherence identified, a validation test suite is built by using a set of *mapping rules*, and is applied against the specification to check whether or not the incoherence in fact leads to an interaction. This is described in Sect. 5. Sect. 6 provides an evaluation of our method.

Briefly, our method identifies situations where interactions are possible, and then formulates tests that should be applied in such situations, to see if interactions really occur.

In principle, formal filtering rules should be determined on the basis of a formal definition of feature interaction. However, because of the fact that whether it is considered that a system presents a feature interaction may depend on the expected behavior of the system from the user's point of view, a universally accepted, formal definition of feature interaction has remained elusive. In this paper, we do not propose such a definition. On the basis of our main idea, we identify six incoherencies that are symptoms of many common known interactions. In Sect. 6, we show that these criteria provide fairly complete coverage with respect to recognized benchmarks.

The examples we use to illustrate the method are those of well established features. However, the method doesn't depend on any specific telephony model and has been used by the authors in order to detect feature interaction in Internet-based telephony systems [9].

It is interesting to note at this point that the feature interaction problem is not limited to telecommunications systems. It exists in software design and [11] shows that it is a concern in the area of building design. We believe that our method can be extended to cover such applications, but this must be left to further work.

It should be noted that this method does not address run-time resolution [2, 3, 12–14] of feature interactions.

2 Definitions and formalism

This section presents the principles and definitions on which the method relies. The description of features is introduced, followed by the definitions for the detection of incoherencies. The rules used for the detection of incoherencies are presented in Sect. 3.

2.1 Features

The evolution of a call process is a sequence of events that bring the system from one state to another. The activation of a feature depends on the state in which the system is and the event that occurs. A feature, in a certain state, is triggered by a specific event: the triggering of this feature leads to the alteration of the call process, according to the functionality provided by the feature. This view is consistent with the view of the Intelligent Network architecture [15] and of Internet telephony based on SIP [16] and CPL [17].

In our method, a feature whose functionality can alter the main call behavior in different states is described in several parts, each part representing the behavior of the feature on a specific state of the call behavior. That is, a feature requires as many description parts as there are different states it affects in the call model.

Formally, a feature X is denoted $\mathcal{F}_X$. A description part is denoted $\mathcal{D}_{X,m}$ where X stands for the name of the feature and m is an integer identifying the description part. For example, a feature X, implemented with three behavioral parts, is broken into three description parts: $\mathcal{D}_{X,1}$, $\mathcal{D}_{X,2}$ and $\mathcal{D}_{X,3}$. The feature $\mathcal{F}_X$ can be stated as $\mathcal{F}_X = \{\mathcal{D}_{X,1}, \mathcal{D}_{X,2}, \mathcal{D}_{X,3}\}$.

Each description part is composed of 4 sets of properties: *pre-conditions*, *triggering events*, *results* and *constraints*.

A property is formally denoted $\gamma(\upsilon_1, \ldots, \upsilon_i)$, $i > 0$, where γ stands for the name of the property and $\upsilon_1, \ldots, \upsilon_i$

is a list of variables representing entities (subscribers, etc.). In order to distinguish names from variables, we follow the Prolog notation where property names start with a lowercase and variable names start with an uppercase.

An example of property is: *talk(A, B)* indicating that some user A talks with some other user B. Another one is: *busy(carol)* stipulating that user *carol* is busy.

Each of the four sets of properties composing a description part is denoted by a letter that identifies its type and is indexed with the name of the feature and the number of the description part. Given a description part $\mathcal{D}_{X,m}$, the sets are denoted as follows:

1. The set of *pre-conditions* describes the state in which the system must be before the activation of the feature can be considered. This set is denoted $\mathcal{P}_{X,m}$. It is an ordered set of properties formally defined as: $\mathcal{P}_{X,m} = \{\gamma_1(\upsilon_{1,1}, \ldots, \upsilon_{1,i}), \ldots, \gamma_j(\upsilon_{j,1}, \ldots, \upsilon_{j,i})\}$, $i, j > 0$, where $\gamma_1, \ldots, \gamma_j$ are properties representing pre-conditions.
   This set must contain the two properties *subs* and *concerns*, denoting the subscriber and the user influenced by the functionality (in many cases, the subscriber itself).
   - *subs(U, F)* – represents a user, denoted by the variable U, that subscribes to a feature, denoted by variable F. For instance, the subscription of user Bob to the feature Call Waiting - represented by the acronym *cw* - is denoted: *subs(bob, cw)*.
   - *concerns(U, F)* – stipulates that the behavior of user U can be influenced by feature F. For instance, Call Waiting implements the possibility, for its subscriber, to hold an incoming call when already involved in another call. Call Waiting also specifies that if someone is held and receives a call, this latter user must be considered as being busy. In this case, the user concerned is not the subscriber but the one on hold. Alice being concerned with Call Waiting is denoted *concerns(alice, cw)*.
2. The set of *triggering events* contains the action(s) triggering the feature. It is an ordered set denoted $\mathcal{T}_{X,m}$, defined as $\mathcal{T}_{X,m} = \{\gamma_1(\upsilon_{1,1}, \ldots, \upsilon_{1,i}), \ldots, \gamma_j(\upsilon_{j,1}, \ldots, \upsilon_{j,i})\}$, $i, j > 0$, where $\gamma_1, \ldots, \gamma_j$ are properties representing triggering events.
3. The set of *results* shows the actions produced by the execution of the feature as well as the state in which the system is after such execution. It is an ordered set denoted $\mathcal{R}_{X,m}$, defined as $\mathcal{R}_{X,m} = \{\gamma_1(\upsilon_{1,1}, \ldots, \upsilon_{1,i}), \ldots, \gamma_j(\upsilon_{j,1}, \ldots, \upsilon_{j,i})\}$, $i, j > 0$, where $\gamma_1, \ldots, \gamma_j$ are properties representing results.
4. The set of *constraints* does not contain properties but rather restrictions on the variables involved in the other sets of properties. This set is denoted $\mathcal{C}_{X,m}$, with $\mathcal{C}_{X,m} = \{(\upsilon_1 \Re_1 \upsilon_2), \ldots, (\upsilon_i \Re_j \upsilon_{i+1})\}$, $i, j \geq 0$, where $\upsilon_1, \ldots, \upsilon_{i+1}$ are the variables used in $\mathcal{P}_{X,m}$, $\mathcal{T}_{X,m}$ and $\mathcal{R}_{X,m}$, and $\Re_1, \ldots, \Re_j$ represent the equality ($=$) or inequality ($\neq$) relations between variables. Such a constraint could be that a caller must be different from the callee (or that they must be one and the same).

As an example, let us consider the feature *Call Forward on Busy*. It provides to the busy subscriber the possibility to forward incoming calls to another party. The properties used for the description of this feature are:

- *busy(B)* – User B is busy.
- *call(A, B)* – Call attempt from a user A to another user B.
- *cfb(C)* – C is the party to which calls are forwarded.
- *redirected(A, C)* – User A is forwarded to user C.

One description part is sufficient to formalize this feature:

$$\begin{aligned}
\mathcal{P}_{cfb,1} &= \{\text{subs(B, cfb), concerns(B, cfb), cfb(C), busy(B)}\}\\
\mathcal{T}_{cfb,1} &= \{\text{call(A, B)}\}\\
\mathcal{R}_{cfb,1} &= \{\text{redirected(A, C), call(A, C)}\}\\
\mathcal{C}_{cfb,1} &= \{(\text{A} \neq \text{B}), (\text{A} \neq \text{C}), (\text{B} \neq \text{C})\}
\end{aligned}$$

The constraints we use here stipulate that values assigned to variables must be unique. This excludes specific cases such as the one where a subscriber calls himself. These constraints can be removed if such cases are to be considered.

Note that it is possible to modify this method to eliminate the distinction between triggering events and pre-conditions. The trigger is an event which, in the presence of pre-conditions, causes the feature to activate. In other words, our method could be easily adapted for the analysis of systems where only a set of pre-conditions leads to some consequences. This would be appropriate for newer features that are triggered by the coexistence of several pre-conditions without the intervention of specific events. However, conventional features have triggering events and their presence simplifies the analysis.

2.2 Incoherencies

As mentioned, incoherencies between sets of properties are symptoms of interactions. Rules are defined to identify such incoherencies. A finite set of constants representing users is considered, and for all bindings of such constants, it is checked if an incoherence arises. Some incoherencies are identified by the fact that different results are specified for the same trigger and precondition. Others are identified because they satisfy explicitly stated contradictions. Note that this is decidable because only finite sets of variables and values are considered, and it is efficient because these sets are small.

Unbound properties are properties expressed using unbound variables. As already mentioned, they are represented in the form: $\gamma(\upsilon_1, \ldots, \upsilon_i)$. To simplify the notation, properties for which variables are bound (have been given a value) are represented without their variables. An unbound property, denoted $\gamma(\upsilon_1, \ldots, \upsilon_i)$ is denoted π when bound. Also, we use the same names $\mathcal{F}_X$, $\mathcal{D}_{X,m}$, etc. to denote sets of properties over variables and over constants.

Contradictory pairs of properties must be identified and listed. A contradiction between two properties is stated using a binary relation denoted $\mathcal{K}$ $(\gamma_1(\upsilon_{1,1}, \ldots, \upsilon_{1,i}), \gamma_2(\upsilon_{2,1},$

…, $\upsilon_{2,j}$)), i, j > 1, where γ_1 and γ_2 are the two properties considered and $\upsilon_{1,i}$, $\upsilon_{2,j}$ are variables that represent users. For instance, a user being idle and busy at the same time is a contradiction. This contradiction is expressed as $\mathcal{K}$(*busy(A)*, *idle(A)*). The use of the same variable A indicates that the two properties are in contradiction when they refer to the same user. This can be similarly expressed by $\mathcal{K}$(*busy(A)*, *idle(B)*) if $A = B$ is a constraint. A data base of contradictions is maintained by the user (see Sect. 4).

2.3 LOTOS concepts

LOTOS [18,19] (Language of Temporal Ordering Specifications) is a Formal Description Technique [20] (FDT) that has been developed within ISO, the International Organization for Standardization. It is based on process algebraic concepts, applying ideas of Milner's CCS [21] and Hoare's CSP [22].

LOTOS represents the behavior of a system by using *behavior expressions* that describe the ordering of *actions*. Actions represent basic behaviors of the system, for example, *off Hook* or *ring*. There is also an internal (invisible) action, corresponding to the τ of CCS, written as **i**. There are basic behavior expressions such as **stop** and process instantiation. Complex behavior expression can be obtained by using operators to combine actions and behavior expressions. Processes are named behavior expressions.

LOTOS includes a basic Abstract Data Type formalism, called ACT ONE, which is used to represent data abstractions. Data can be associated with actions in two ways: **!**, meaning value offer, and **?**, meaning value query. These can be combined in actions:

```
switch !subs ?dest:dest_sort
```

denotes an action on gate *switch* where the current value of the identifier *subscriber* is offered, and a value for *destination* is queried simultaneously.

Main operators are ;, defining sequencing of actions in a behavior expression, |||, defining the situation where the actions of two behavior expressions interleave, ||, defining the situation where the actions of two behavior expressions must execute together step by step, and |[$a_1, \ldots, a_n$]|, defining the situation where the actions $a_1, \ldots, a_n$ must execute together, while all other actions interleave.

LOTOS has two main assets for specifying telephony systems and their features: it is capable of representing clearly and precisely system structures, and it has a good set of animation and validation tools. Animation tools allow the application of test scenarios (scenario based validation). They allow the designer to use various techniques to detect feature interactions that can be present in the specification of a system.

In this paper, only a small subset of LOTOS is used and therefore a complete description of the language is not given. Sufficient comments are included in Sect. 5.3 to make the paper self-contained.

3 Filtering rules

This section presents our rules for the detection of incoherencies. For space considerations, rules are presented in a formal way with only a few examples. More complete explanations and examples for each rule can be found in [23].

Our classification of incoherencies uses the following principles. First, we differentiate between direct (denoted by the letter D below) and transitive (denoted by the letter T) incoherencies. Direct incoherencies are between simultaneously activated features and transitive incoherencies are between sequentially activated ones. Direct incoherencies are further classified as to whether the features relate to one or different subscribers (numbers 1 and 2, resp.), and then again as to whether they present a simple non-determinism or an explicitly identified contradiction (letters a and b, resp.). Recapitulating:

Rules D1-a and D1-b – identify direct incoherencies between two features having one subscriber and presenting incoherent results, without a contradiction or with a contradiction.

Rules D2-a and D2-b – identify direct incoherencies between two features having different subscribers and presenting incoherent results, without a contradiction or with a contradiction.

Rules T1 and T2 – identify transitive incoherencies between two features where one triggers the other and the results are incoherent or lead to a loop.

In order to formalize the identification of incoherencies, a few definitions need to be stated:

- The combination of two feature descriptions is defined as the pair of the two descriptions, where the variables of a description are considered to be distinct from the ones of the other description, even if they have the same names. Given two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$, we formally denote their combination $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$.
- A user cannot subscribe to a feature more than once. This is expressed as a general contradiction denoted $\mathcal{K}$(subs(A, X), subs(A, X)) where the variable A represents a user and variable X represents the name of a feature.
- $\mathcal{S}(\mathcal{C}_{X,m})$ denotes that all constraints of the constraint set $\mathcal{C}_{X,m}$ are satisfied.
- Υ is the finite set of users that can be used to bind user variables in properties.
- Two distinct ordered sets $\mathcal{E}_1$ and $\mathcal{E}_2$ present a contradiction if there exists a $\pi_1 \in \mathcal{E}_1$ and a $\pi_2 \in \mathcal{E}_2$ such that $\mathcal{K}(\pi_1, \pi_2)$ is satisfied.

3.1 Direct incoherence rules

Direct incoherencies occur when two description parts $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ act on the same triggers but lead to different or contradictory results. They are symptoms of non-determinism.

3.1.1 Direct incoherence rule D1-a

This rule identifies incoherencies between features subscribed by one user, triggered by the same event and yielding different but not contradictory results. Let us consider two feature description parts $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ combined in a pair $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$. An incoherence is present if there exists a binding of variables from Υ, the set of all users, in the combination of the two features, such that the following holds:

1. $\exists\, \upsilon \in \Upsilon \mid (\text{subs}(\upsilon, X) \in \mathcal{P}_{X,m} \wedge \text{subs}(\upsilon, Y) \in \mathcal{P}_{Y,n})$
   – User υ subscribes to features X and Y –
2. $\nexists(\pi_1 \in \mathcal{P}_{X,m}, \pi_2 \in \mathcal{P}_{Y,n}) \mid \mathcal{K}(\pi_1, \pi_2)$
   – No contradictory pre-conditions –
3. $\mathcal{T}_{X,m} = \mathcal{T}_{Y,n}$
   – Same triggers –
4. $\mathcal{R}_{X,m} \neq \mathcal{R}_{Y,n}$
   – Different results –
5. $\nexists(\pi_3 \in \mathcal{R}_{X,m}, \pi_4 \in \mathcal{R}_{Y,n}) \mid \mathcal{K}(\pi_3, \pi_4)$
   – No contradictory results –
6. $\mathcal{S}(\mathcal{C}_{X,m}) \wedge \mathcal{S}(\mathcal{C}_{Y,n})$
   – All constraints are satisfied –

To illustrate the mechanism, let us consider two features: *Call Forward on Busy* (already presented in Sect. 2.1) and *Incoming Call Screening*. As mentioned, *Call Forward on Busy* provides to the busy subscriber the possibility to forward incoming calls to another party. *Incoming Call Screening* allows the subscriber to establish a screening list and to deny calls incoming from users in the list. The properties used in the description of these two features are:

- *busy(B)* – User B is busy.
- *call(A, B)* – Call attempt from a user A to another user B.
- *cfb(C)* – C is the party to which calls are forwarded.
- *redirected(A, C)* – User A is forwarded to user C.
- *ics_list(A)* – User A is in the screening list of the subscriber.
- *deny_call(B, A)* – Calls incoming from A are denied by user B.
- *call_denied(B, A)* – User B notifies user A that the call has been denied.

Each one of these features can be formalized in one description part only, so the subscript 1 below denotes the (single) description part of each feature. The variables in the descriptions are restricted to different values, to exclude cases such as the one where a user forwards incoming calls to himself when he is busy.

The acronym *cfb* denotes *Call Forward on Busy* and *ics* denotes *Incoming Call Screening*:

- *Call Forward on Busy* part 1, subscriber B is busy and receives a call from A:

$$\begin{aligned}
\mathcal{P}_{cfb,1} &= \{\text{subs(B, cfb), concerns(B, cfb), cfb(C),}\\ &\qquad \text{busy(B)}\}\\
\mathcal{T}_{cfb,1} &= \{\text{call(A, B)}\}\\
\mathcal{R}_{cfb,1} &= \{\text{redirected(A, C), call(A, C)}\}\\
\mathcal{C}_{cfb,1} &= \{(\text{A} \neq \text{B}), (\text{A} \neq \text{C}), (\text{B} \neq \text{C})\}
\end{aligned}$$

- *Incoming Call Screening*, part 1, screened party C calls B:

$$\begin{aligned}
\mathcal{P}_{ics,1} &= \{\text{subs(B, ics), concerns(B, ics), ics\_list(A)}\}\\
\mathcal{T}_{ics,1} &= \{\text{call(A, B)}\}\\
\mathcal{R}_{ics,1} &= \{\text{deny\_call(B, A), call\_denied(B, A)}\}\\
\mathcal{C}_{ics,1} &= \{(\text{A} \neq \text{B})\}
\end{aligned}$$

Performing pair-wise combination over $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ leads to observe that $|\mathcal{D}_{cfb,1} \bullet \mathcal{D}_{ics,1}|$ presents an incoherence since it is possible to find a binding of variables that satisfies direct incoherence rule D1-a.

Let us use the notation $X \leftarrow val$ to denote the binding of value val to variable X and consider three users: *Alice*, *Bob*, *Carol*. One of the possible bindings of the variables of the properties of $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ is as follows:

- $\mathcal{D}_{cfb,1}$: $A \leftarrow$ alice, $B \leftarrow$ bob, $C \leftarrow$ carol.

$$\begin{aligned}
\mathcal{P}_{cfb,1} &= \{\text{subs(bob, cfb), concerns(bob, cfb),}\\ &\qquad \text{cfb(carol), busy(bob)}\}\\
\mathcal{T}_{cfb,1} &= \{\text{call(alice, bob)}\}\\
\mathcal{R}_{cfb,1} &= \{\text{redirected(alice, carol),}\\ &\qquad \text{call(alice, carol)}\}^{\alpha}\\
\mathcal{C}_{cfb,1} &= \{(\text{alice} \neq \text{bob}), (\text{alice} \neq \text{carol}),\\ &\qquad (\text{bob} \neq \text{carol})\}
\end{aligned}$$

- $\mathcal{D}_{ics,1}$: $A \leftarrow$ alice, $B \leftarrow$ bob.

$$\begin{aligned}
\mathcal{P}_{ics,1} &= \{\text{subs(bob, ics), concerns(bob, ics),}\\ &\qquad \text{ics\_list(alice)}\}\\
\mathcal{T}_{ics,1} &= \{\text{call(alice, bob)}\}\\
\mathcal{R}_{ics,1} &= \{\text{deny\_call(bob, alice),}\\ &\qquad \text{call\_denied(bob, alice)}\}^{\beta}\\
\mathcal{C}_{ics,1} &= \{(\text{alice} \neq \text{bob})\}
\end{aligned}$$

If Alice calls Bob (call(alice, bob)), *Call Forward on Busy* should be activated and Alice should be redirected to Carol and call Carol (α). At the same time, *Incoming Call Screening* should be activated and Alice's call should be denied (β).

We conclude that $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ present an incoherence characterized by non-determinism. Should the call be redirected or denied ?

3.1.2 Direct incoherence rule D1-b

Direct incoherence rule D1-b identifies incoherencies that result in explicit contradictions. Similarly to rule D1-a, this rule identifies *symmetric* incoherencies. The clauses we use for this rule are the same as for rule D1-a except that clauses 4 and 5 are replaced by a single one:

4. $\exists(\pi_3 \in \mathcal{R}_{X,m}, \pi_4 \in \mathcal{R}_{Y,n}) \mid \mathcal{K}(\pi_3, \pi_4)$

An example of interaction that is detected by this rule is the following. A trivial rule is that a call cannot be at the same time held and denied. This rule should be part of the data base of contradictions. Suppose now that Bob has Call Waiting (CW) and Incoming Call Screening (ICS) with Alice on the ICS list. Now if Alice calls Bob, Call Waiting may be activated, and hold Alice. But because of ICS, Alice's call should also be denied. This violates the general rule. Binding the variables in the features descriptions to these user names produces the contradiction. If not properly addressed, this interaction could lead to Alice being connected to Bob.

3.1.3 Direct incoherence rule D2-a

Rule D2-a is very similar to rule D1-a in the sense that it identifies incoherencies taking place between features when both features can be triggered by the same event and present different results. The difference is that this rule considers features subscribed by different users and both features concern the subscriber of one of the features. Thus, we use the same clauses as for rule D1-a except for the first one which is replaced by:

1. $\exists\, \upsilon_1, \upsilon_2 \in \Upsilon, \upsilon_1 \neq \upsilon_2 \mid$
   $(\text{subs}(\upsilon_1, X) \in \mathcal{P}_{X,m} \wedge \text{concerns}(\upsilon_2, X) \in \mathcal{P}_{X,m})$
   $\wedge$
   $(\text{subs}(\upsilon_2, Y) \in \mathcal{P}_{Y,n} \wedge \text{concerns}(\upsilon_2, Y) \in \mathcal{P}_{Y,n})$

Due to its configuration, this rule does not identify *symmetric* incoherencies. As an example of interaction that is detected by this rule, consider the Call Waiting (CW) and Call Forward on Busy (CFB) features. They have the same preconditions, the same triggers, but different results. Suppose that Alice subscribes to CW, and Carol to CFB. Carol is busy and held by Alice. When Carol receives a call, should the caller receive a busy message or be forwarded ?

3.1.4 Direct incoherence rule D2-b

Similarly to direct incoherence rule D2-a, rule D2-b identifies incoherencies present between features subscribed by different users and concerning the same user in the case where both features are triggered by the same triggering event and present different results. But this rule is also similar to D1-b since it is characterized by contradictory results. Also, as for D2-a, this rule does not identify *symmetric* incoherencies.

We use the same clauses as for rule D2-a except that clauses 4 and 5 are replaced by a single one. We are looking for contradictions, thus, the clause used here is the same one as for rule D1-b.

4. $\exists(\pi_3 \in \mathcal{R}_{X,m}, \pi_4 \in \mathcal{R}_{Y,n}) \mid \mathcal{K}(\pi_3, \pi_4)$

An example of interaction detected by this rule is a second type of interaction between Call Waiting and Incoming Call Screening. This interaction can be detected if the database of contradictions contains the general rule: *An incoming call cannot be denied and receive a busy signal at the same time.* Suppose that Alice subscribes to CW and Carol to ICS, with Dave on the list. Carol is busy and held by Alice. What treatment should Dave receive if it calls Carol ?

3.2 Transitive incoherence rules

Transitive incoherencies can occur when the results of one feature trigger a second feature, and there is no contradiction between the pre-conditions of the two features (in other words, the and-composition of the two preconditions is satisfiable).

Two possible incoherencies are identified in this situation: The results of the two descriptions present a contradiction, or the results of a description trigger the other one and vice-versa (which results in a loop). Contradictions and loops are identified by transitive incoherence rules T1 and T2, respectively.

The transitivity relation is an implication relation. The results of the feature $\mathcal{D}_{X,m}$ trigger the feature $\mathcal{D}_{Y,n}$. In other words, we can stipulate that $\mathcal{D}_{X,m}$ *implies* $\mathcal{D}_{Y,n}$.

3.2.1 Transitive incoherence rule T1

Transitive rule T1 identifies incoherencies caused by the results of a feature triggering another feature that has results presenting a contradiction with respect to the results of the first feature. Formally, this is defined by the following clauses:

1. $\nexists(\pi_1 \in \mathcal{P}_{X,m}, \pi_2 \in \mathcal{P}_{Y,n}) \mid \mathcal{K}(\pi_1, \pi_2)$
2. $\mathcal{R}_{X,m} \supseteq \mathcal{T}_{Y,n}$
3. $\exists(\pi_3 \in \mathcal{R}_{X,m}, \pi_4 \in \mathcal{R}_{Y,n}) \mid \mathcal{K}(\pi_3, \pi_4)$
4. $\mathcal{S}(\mathcal{C}_{X,m}) \wedge \mathcal{S}(\mathcal{C}_{Y,n})$

As an example, let us consider *Outgoing Call Screening* (OCS) and combine it with the feature *Call Forward on Busy* (CFB). OCS allows the subscriber to establish a screening list and block at the originating point calls made to users that are in this list.

A general rule that should be in the contradiction data base is that a call cannot be performed and blocked at the same time:

– $\mathcal{K}(\text{call(A, B)}, \text{block\_call(A, B)})$

If Alice (who has OCS to block calls to Carol) calls Bob (who is busy and has CFB to Carol), CFB is triggered and the call goes to Carol. This call should be performed because of CFB, but is blocked because of OCS. This is a contradiction.

3.2.2 Transitive incoherence rule T2

Transitive incoherence rule T2 identifies incoherencies characterized by loops. Two features $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ enter a loop when $\mathcal{D}_{X,m}$ implies $\mathcal{D}_{Y,n}$ and vice-versa. Whether their results present a contradiction or not is not checked since the goal is to identify loops caused by transitivity of features, not contradictions, which are identified already by the rule T1. For this rule, we use the same clauses as for rule T1 except that clause 3 is replaced by the following:

3. $\mathcal{R}_{Y,n} \supseteq \mathcal{T}_{X,m}$

A well-known example of interaction detected by this rule is the interaction of call forward with itself, which can lead to forwarding loops.

4 Automatic filtering using prolog

Given n feature descriptions to analyze, pair-wise combination leads to n^2 possible pairs. We proposed 6 rules for incoherence detection. Hence, probing each pair with each one of the 6 rules leads to $6n^2$ cases to analyze. We represent features and rules as logic predicates. We have implemented a solution to carry out this process using Prolog [24] and the interpreter used was SWI Prolog [25].

4.1 Features

The implementation of feature descriptions in Prolog follows a simple translation scheme. Properties are represented as facts. Ordered sets of properties are represented as lists of facts. A feature description is represented as a predicate called *feature*. The signature of this predicate is *feature(N, P, T, R) :- C*, where variables *N*, *P*, *T*, *R* represent, respectively, the lists containing name, pre-conditions, triggering events and results, and variable *C* represents the body containing the constraints.

In order to be able to bind all variables contained in the properties of a pair of feature descriptions to different values (user names), a number of values up to the sum of variables of both descriptions may be required. Each user is defined using a Prolog fact *user* containing the name of the user. Thus, defining four users for the system is done by defining four *user* facts, each representing a different user.

As an example, the Prolog representation of *Call Forward on Busy*, given in Sect. 3.1.1, is the following:

```
feature([cfb, 1],
        [subs(B, cfb), concerns(B, cfb),
         cfb(C), busy(B)],
        [call(A, B)],
        [redirected(A, C), call(A, C)]
   ):-
        user(A), user(B), user(C),
        A \= B, A \= C, B \= C.
```

When the predicate *feature* is called, Prolog binds variables *A*, *B* and *C* to values taken from the *user* facts in such a way that relations between variables (constraints) are satisfied. The last line states that all users must be different.

The *contradiction* predicate states a contradiction between two facts. It represents the predicate $\mathcal{K}$ defined in Sect. 2.2. As mentioned, an example of a contradiction is that a user *X* cannot be busy and idle at the same time. This is expressed by: *contradiction(busy(X), idle(X))*. In this way, the Prolog program implements the data base of contradictions.

4.2 Rules

Rules are implemented in the form of Prolog predicates:

```
fi_check(rule_name,[F1,x],[F2,Fy],PcnF1,
     TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
```

The variables denote, respectively: the name of the rule, the names (split in two parts) of the two features to be considered, the pre-conditions, the triggering events and results of the first feature, and the pre-conditions, triggering events and results of the second one. Rule D1-a, given in Sect. 3.1.1, is:

```
fi_check(d1_a,[F1, Fx], [F2, Fy], PcnF1,
    TrgF1, ResF1, PcnF2, TrgF2, ResF2):-

    feature([F1, Fx],
            PcnF1, TrgF1, ResF1), !,
    feature([F2, Fy],
            PcnF2, TrgF2, ResF2),
    member(subs(U, F1), PcnF1),
    member(subs(U, F2), PcnF2),
    not(contradiction(PcnF1, PcnF2)),
    TrgF1 = TrgF2,
    ResF1 \= ResF2,
    not(contradiction(ResF1, ResF2)), !.
```

Prolog's unification mechanism binds variables in order to try and satisfy predicates and facts. Once a successful binding is found, Prolog may be asked to search for other bindings that might also lead to solutions.

4.3 Incoherency reports

Each incoherency report is composed of two parts: information about the incoherence and the incoherence itself. The information about the incoherence is presented in plain English in a format tailored to the rule concerned. It contains the name of the rule, a brief description of the type of incoherence, as well as information about the incoherence, all written as a Prolog comment. An example of this will be given in Sect. 5.3. The incoherence itself is reported by the tool as a fact called *fi* containing the description of the features involved. This fact contains nine parameters (lines 1 to 9): the rule used (line 1), the names of the two features involved (lines 2 and 3), the pre-conditions (line 4 and 7), the triggering events (lines 5 and 8), and the results (lines 6 and 9) of both features. As an example, the report for the incoherence between $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ (Sect. 3.1.1) is:

```
01: fi(d1,
02:     [cfb,1],
03:     [ics,1],
04:     [subs(bob,cfb),concerns(bob,cfb),
         cfb(carol), busy(bob)],
05:     [call(alice, bob)],
06:     [redirected (alice, carol),
         call(alice, carol)],
07:     [subs(bob,ics),concerns(bob,ics),
         ics_list(alice)],
08:     [call(alice, bob)],
09:     [deny_call (bob, alice),
         call_denied(bob, alice)]
10: ).
```

4.4 Algorithm

For a given rule r and a given combination of features $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$, the analysis deals with the properties and variables of feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$. As an example, let us consider two features $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ analyzed by rule T1. The analysis is done as follows:

```
1. Bind variables of $\mathcal{D}_{X,m}$ to values such that $\mathcal{C}_{Y,n}$ is
   satisfied and go to step 2.

2. Value combinations remain untried ?
   Yes. Bind variables of properties of $\mathcal{D}_{Y,n}$ to first possible
        combination that satisfies $\mathcal{C}_{X,m}$ and go to step 3.
   No.  Go to step 7.

3. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ present a contradiction ?
   Yes. Backtrack to step 2.
   No.  Go to step 4.

4. $\mathcal{R}_{X,m} \supseteq \mathcal{T}_{Y,n}$, or, $\mathcal{R}_{X,m} = \mathcal{T}_{Y,n}$ ?
   Yes. Go to step 5.
   No.  Backtrack to step 2.

5. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ present a contradiction ?
   Yes. Go to step 6.
   No.  Backtrack to step 2.

6. The rule is satisfied, an incoherence is identified.

7. Rule fails, no incoherence is identified.
```

Now, let us consider a set of new features $\mathcal{T}$, and the new set of all features $\mathcal{U} = \mathcal{S} \cup \mathcal{T}$. The list of all combinations is obtained by $\mathcal{U}$: $\mathcal{S} \cup \mathcal{T} \times \mathcal{S} \cup \mathcal{T}$. This means that pair-wise combinations already analyzed are considered again, while they need not be since the result is already known. This is avoidable:

- The first solution consists in keeping the set of previously detected incoherencies in a data base, such that for a given rule, only pairs not known as presenting incoherencies are considered. However, this only gives information about known incoherencies. Other pairs, already analyzed but not presenting any incoherence, are processed again.
- The second solution is to consider two different sets $\mathcal{S}$ and $\mathcal{T}$ and to determine the pairs to be considered using a relation on $\mathcal{S} \times \mathcal{T}$. For instance, if a new feature description $\mathcal{D}_{X,m}$ is added, one can specify two sets such that $\mathcal{S}$ contains all features of the old specification and $\mathcal{T}$ contains the newly added feature. Then, $\mathcal{S} \times \mathcal{T}$ determines all pairs containing the new feature. Moreover, all pairs previously analyzed may be ignored.

Depending on the size of the specification and on the number of feature descriptions to be analyzed, reusing already known incoherencies, automatically obtained from reports, can take less time than building and managing the two sets of features to be considered. The availability of the two solutions allows the user to choose the best one.

5 Automatic test suite generation

The identification of incoherencies, discussed so far, is a filtering method. The representation method used in this phase was devised for rapid detection and includes only the high-level representation aspects of a feature. Since the filtering information is obtained from feature descriptions, identifying an incoherence at the filtering stage does not imply that the corresponding interaction really exists. The behavioral specification of the features may already contain the fix for the interaction.

Hence, the technique used to determine whether a possible interaction detected by filtering really exists in the specification consists of generating corresponding test scenarios and exercising them against the specification.These scenarios are generated automatically from the reports obtained in the filtering phase. They can be seen as test cases which, if successful, show that the interaction can actually occur in the system according to the behavior specifications. Automatic generation of scenarios was made possible by the definition of rules mapping properties to scenario parts.

Note that this section presents the generation of scenarios based on the LOTOS [18, 19] specification we studied in work with the company Mitel.

5.1 Scenarios generation principles

Properties used to describe a feature define the state in which the feature must be, the event(s) triggering the feature and the results produced by the activation of this feature. The mapping to go from a feature interaction report to a test scenario is done by associating each property in the report to a sequence of actions in the LOTOS specification. To generate a test scenario, our method uses four predicates. The body of these predicates needs to be programmed by the tester to implement scenario generation. The predicates are:

test_header – The LOTOS specification implements a database that allows dynamic specification of feature attributes (e.g. which user to screen, where to call forward). These attributes refer to the subscriber as well as the users concerned with the feature. This database must be initialized with proper values at the beginning of each test scenario. The *test_header* predicate is used to produce the LOTOS test header and to initialize the database. The initialization of the database is constructed using a sub-predicate *lotos_database* that builds the initialization set.

test_pre_conditions – This predicate is used to decompose the list of pre-condition properties and to produce the LOTOS event(s) relative to each property. The decomposition is done sequentially in the same order as the properties. The corresponding LOTOS action(s) are produced by calling a sub-predicate containing the action(s) to produce.

test_triggers – This predicate is built on the same principles as the previous one and uses the sub-predicate *test_trigger* to produce sequences of LOTOS actions corresponding to each property.

test_results – This predicate is built on the same principles as the previous one.

Pre-conditions are mapped using *test_pre_condition*, triggering events are mapped using *test_triggering_event* and results are mapped using the predicate *test_results*. Such a partitioning allows to specify different mappings for the same property, depending on the set it belongs to in the feature description. Additional predicates, for example predicates to map users to phone numbers, can be defined.

Note that the mapping rules are specified once only, for the whole set of features. Because of the fact that these predicates need to be programmed, our test generation method can be considered semi-automatic, however it is automatic once the programming is done.

5.2 Test suite principles

We have seen that an interaction, corresponding to an incoherence identified in the filtering process, can be characterized as non-determinism, a contradiction (which is also a sort of non-determinism) or a loop. A testing model corresponding to each of the three possible kinds of interactions is defined. The models defined are then used for the derivation of test scenarios and analysis of results.

Non-determinism – Non-determinism implies that the result is ambiguous. Interaction detection is done using a test suite composed of two scenarios. Both scenarios model cases where both features are present. The first scenario illustrates the case where the results of the first feature occur and the second one illustrates the case where the results of the second feature occur. The result of the application of both scenarios may be one of the following:

- Both scenarios are unsuccessful. This definitely indicates a problem. It tells the designer that in any case, neither feature behaves properly.
- Both scenarios are successful. This means that both results can occur. This indicates a problem in the specification because only one feature at a time should be activated. However, depending on the features, this result can be acceptable. It is up to the designer to ensure that the behavior is the one intended.
- One scenario is successful and the other is not. This indicates that no non-determinism exists. However, the behavior may not be the one intended. If the designer expects a different result than the one observed, then a problem is present.

Contradictions – A contradiction implies non-determinism. The results of the two features are in contradiction, and so the result is not known. Thus, scenarios for this kind of interaction follow the same model as the previous one.

Loops – A loop can either occur or not occur. Then, only one scenario is needed to detect such interactions. The scenario illustrates the case where the loop occurs, rejection of the scenario indicates that no loop occurs, and success of the scenario indicates that a loop occurs in the specification.

A designer having knowledge of the models used can interpret the results of the application of scenarios and give a verdict. It should be noted that the abstract test scenarios that are generated at this stage can be re-used after implementation to test for feature interactions at that stage.

5.3 Example of use

Let us consider the incoherence identified by rule D1-b for Call Forward Always and Call Forward on Busy. This incoherence is due to the fact that if a user A subscribes to both features and forwards his calls to some user X with the first feature and to some user Y with the other feature, it is not clear what should happen if A is called when he is busy. The trigger for both features is present and leads to contradictory results. The report for this incoherence follows.

```
% * Rule D1-b -> [cfb, 1] & [cfa, 1]

% A user subscribes to two features
% having the same triggering events and
% contradictory results

%    + Features pre-conditions
%       - Pre-conditions of [cfb, 1]
%           subs(bob, cfb)
%           concerns(bob, cfb)
%           cfb(carol)
%           busy(bob)

%       - Pre-conditions of [cfa, 1]
%           subs(bob, cfa)
%           concerns(bob, cfa)
%           cfa(dave)

%    + Same triggering events

%         call(alice, bob)
%    + Contradictory results

%       - Resulting events of [cfb, 1]
%           redirected(alice, carol)
%           call(alice, carol)
%           ring(alice, carol)

%       - Resulting events of [cfa, 1]
%           redirected(alice, dave)
%           call(alice, dave)
%           ring(alice, dave)
```

The corresponding derived test suite consists of two scenarios: the first one where Call Forward on Busy is triggered and the second one where Call Forward Always is triggered.

As an example, one of the two LOTOS test processes produced by our tool (resulting in Call Forward on Busy) is presented. We first provide some explanatory remarks about the system's architecture to enable the reader to better understand the scenario.

The system in consideration is a new experimental architecture for Internet telephony being designed by the company Mitel. Call processing goes through three types of components (agents):

Device Agents (DAGENT): handle the devices, i.e. the physical endpoint of a call (e.g. telephone, a computer capable of voice over IP, diary, etc.).

Personal Agents (PAGENT): know the restrictions and privileges given to specific users. They also know the preference relationships between a user and his/her devices.

Functional Agents (FAGENT): represent the functional role of the endpoint of a call (e.g. president, director, secretary, etc.).

Many instances of DAGENTs, PAGENTs and FAGENTs can be active simultaneously for one user. From a scenario perspective, it is also helpful to distinguish between the roles played by these components (e.g. originating or terminating). The features are processed in the components as appropriate.

The exact functionalities of these components are not important for the understanding of the methodology and thus will not be described further.

The scenario is a process simulating the interaction of users with the architecture. Users are identified by a name (`user_A`, `user_B`, `user_C`). The associated devices are identified by `da_` followed by the ID of the user and the instance of the device. Communication between users and their respective devices is performed using the LOTOS synchronization principles presented earlier. Sending a message from a user to a device agent thus consists of the action `USER_to_DA` followed by value offers. The first two respectively represent the ID of the user and the ID of the Device. Remaining value offers represent the message sent. For instance, going *offHook* consists of the following LOTOS action:

```
USER_to_DA !user_B !da_B0 !offHook;
```

This is a communication between a user and a Device Agent. It involves user_B and the device agent 0 of this user. The message sent is `offHook` stipulating that the user is going offHook. The complete test scenario generated by the tool is:

```
process
  (* name of the scenario *)
  ScenarioFI_RD1b_cfb1_cfa1_rslt_in_cfb1
   [

    (* communication gates with system*)
    USER_to_DE, DE_to_USER, Init,success

   ]: noexit:=

  (* database initialization *)
  (* programmation of features CFA and CFB *)
  (* with their parameters and subscribers *)
  let specificDB:Database =

  (*User B subscribes to CFB and calls*)
  (* are redirected to number 2003*)
  UF(userB, FD(cfb, fArg(2003), noArg),
  endFSet),

  (*User B subscribes to CFA and calls*)
  (* are redirected to number 2004 *)
  UF(userB, FD(cfa, fArg(2004), noArg), endFSet),
  emptyDB)))) in (

    (* Initialization of the database *)
    Init !specificDB;
    (* user B is going offHook *)
    USER_to_DA !user_B !da_B0 !offHook;
    (* and receives a dialTone *)
    DA_to_USER !da_B0 !user_B !dialTone;

   (* user A is going offHook *)
   USER_to_DA !user_A !da_A0 !offHook;
   (* receives a dialTone *)
   DA_to_USER !da_A0 !user_A !dialTone;
   (* dials the number of user B *)
```

```
    USER_to_DA!user_A !da_A0 !dial !2002;
    (*and receives progress notification*)
    DA_to_USER!da_A0!user_A!callIn Progress;

    (*receives redirection notification*)
    DA_to_USER!da_A0!user_A!redirectTone;

    (* user C receives a ringing tone *)
    DA_to_USER !da_C0 !user_C !ringingOn;

    (* successful end of scenario *)
    success; stop
  )
endproc
```

The tool LOLA [26] was used to execute the LOTOS tests and report the results. The application of the two test scenarios using LOLA allows the designer to verify if:

– First, only one of the two features is always triggered since, when applied to the specification using LOLA, one of the test scenarios results in a MUST PASS while the other results in a REJECT.
– Second, the feature that is triggered is the intended one. This must be verified by the designer, in accordance with the test results (scenario resulting in a MUST PASS) and the requirements.

For our detailed specification, the answer is that Call Forward Always is the feature that is triggered when both features are present, whether the user is busy or not. The designer may wish to change this situation, or to allow it. In this second case, the user should be discouraged from subscribing to Call Forward on Busy together with Call Forward Always.

The specification could have been written in a way as to allow both features to be triggered simultaneously upon reception of a call. This possibility leads to race conditions and unpredictable behavior, obviously a disastrous design error that our tool allows to detect (MAY PASS for both test scenarios).

6 Application and results

The approach was applied to two case studies: one provided by the company Mitel, already discussed, and another developed for a contest held in the occasion of the Feature Interaction Workshop 2000 [3]. The Mitel case study went through all phases of the process. The Feature Interaction Workshop case study went through the filtering phase only. For this reason, only the results of the Mitel case study are presented here, however the results for the second case study can be found in [23].

6.1 Application

The approach applied in the Mitel project consists of expressing requirements with a Use Case Maps model [12, 27], deriving a LOTOS specification corresponding to the UCM model and validating the LOTOS specification. Mitel Corp. provided us with the UCM model, from which a LOTOS specification was derived.

Using Use Case Maps and additional documentation provided by Mitel Corp., the features were modeled and our filtering method was applied.

These features considered were: Outgoing Call Screening (OCS), Incoming Call Screening (ICS), Call Forward Always (CFA), Call Forward on Busy (CFB), Call Transfer (CT), Call Pickup (CP), Call Waiting (CW), Automatic Recall (AR) and Time Reminder (TR). These features are presented in [27] as well as in [23].

Some of the features are represented using a single description while others need up to three or four. Call Waiting is split in four parts. Call Transfer and Automatic Recall are split in three parts. Time Reminder is split in two parts. The other features are represented using only one description. In total, we have twenty feature descriptions.

As an example of features definition, let us consider Call Forward Always and Call Forward on Busy. Both features are defined using the following common properties:

– `call(A, B)` – Call attempt from user A to user B.
– `redirected(A, C)` – User A is forwarded to user C.
– `ring(A, C)` – User A rings user C.

Call Forward Always uses the additional properties:

– `subs(A, cfa)` – User A subscribes to Call Forward Always.
– `concerns(A, cfa)` – Call Forward Always concerns user A.
– `cfa(C)` – Calls are forwarded to user C.

And Call Forward on Busy uses the additional following ones:

– `subs(A, cfb)` – User A subscribes to Call Forward on Busy.
– `concerns(A, cfb)` – Call Forward on Busy concerns user A.
– `busy(A)` – User A is busy.
– `cfb(C)` – Calls are forwarded to C.

Each feature only implements one functionality, hence only one *feature* predicate is needed for each one of them. The Prolog implementation of features follows:

– Call Forward Always – A call from user A to user B is forwarded to the specified destination (user C), as stated by the property $cfa(C)$.

```
feature([cfa, 1],
        [subs(B, cfa),
         concerns(B, cfa), cfa(C)],
        [call(A, B)],
        [redirected(A, C),
         call(A, C), ring(A, C)]
     ):-
        user(A),
        user(B), A \= B,
        user(C), C \= A, C \= B.
```

– Call Forward on Busy – When user B is busy, a call from user A to user B is forwarded to the specified destination (user C), as stated by the property $cfb(C)$.

```
feature([cfb, 1],
        [subs(B,cfb),concerns(B,cfb),
         cfb(C), busy(B)],
        [call(A, B)],
        [redirected(A, C),call(A, C),
         ring(A, C)]
    ):-
        user(A),
        user(B), A \= B,
        user(C), C \= A, C \= B.
```

In addition, we consider that a user A making a call to a user B is in contradiction with the same user A making a call to a different user C. This is stated as:

– $\mathcal{K}$(`call(A, B),call(A, C)`) if and only if $B \neq C$.

This is implemented in Prolog using the predicate *contradiction* in the following form:

– `contradiction(call(A,B), call(A,C)):- B \= C.`

6.2 Results

Our filtering method identified a total of 43 incoherencies. Table 1 presents the results obtained for the complete analysis. It illustrates direct and transitive incoherencies.

Table 2 presents a statistical summary of the incoherencies detected. Note that incoherencies identified by rules D1-a, D1-b, T1 and T2 are easier to foresee and resolve. Incoherencies identified by rules D2-a and D2-b are much more complex. As illustrated by Table 2, only a few incoherencies are identified by rules D2-a and D2-b. This thus indicates that the designers mostly have to deal with simple incoherencies.

As explained in Sect. 5, the tool produces test suites composed of one or two test scenarios. Scenarios are produced using the predicates presented there, plus others to be defined as required. The LOTOS specification already existed for 6 of the features in consideration (OCS, ICS, CFA, CFB, CT, CP) when the feature descriptions were formalized and the tool built. Test suites were derived and tested against the LOTOS specification for these features. In the end no real interactions were found among those 6 features: clearly, they were well known to the specifiers, and interactions were avoided by the Mitel designers at the detailed design stage. However, all incoherencies identified had the potential to lead to feature interactions.

This process presents two limitations. First, although the principles presented in this paper also apply to the detection of multi-way interactions, this extension requires further work, especially from the point of view of efficiency. Hence, only pair-wise combinations are considered, thus restricting the detection to two-way interactions. Multi-way interactions involving features such as three way calling cannot be considered for analysis. Second, the interactions detection mechanism is based on a systematic pair-wise combination of features, thus resulting in polynomial execution time.

Table 1 Filtering Results

	Incoherencies Detected								
	OCS	ICS	CFA	CFB	CT	CP	CW	AR	TR
OCS			1	1	2	1		1	
ICS	–		2	2	1	1	3	2	
CFA	–	–	2	4	1	1	3	2	
CFB	–	–	–	2	1		3	2	
CT	–	–	–	–			1		
CP	–	–	–	–	–				
CW	–	–	–	–	–	–	1	3	
AR	–	–	–	–	–	–	–		
TR	–	–	–	–	–	–	–	–	

Table 2 Summary of Results

Incoherencies per rule and type						
Rule	D1-a	D1-b	D2-a	D2-b	T1	T2
Totals	13	3	4	1	19	3
Per rule	30%	7%	9.5%	2.5%	44%	7%
Per type	49%				51%	

7 Conclusion & future research

We have proposed an approach for feature interaction detection. It uses two feature descriptions: one that formalizes the feature requirements, and the other that formalizes the detailed feature behavior. It consists of a filtering stage that allows the automatic identification of incoherencies between requirements, and a testing stage that checks whether the potential interactions identified by incoherencies exist in the detailed behavior.

The filtering process is based on rules characterizing incoherencies. Our tool performs pair-wise analysis of features and identifies pairs that satisfy one or more incoherence rules.

The test suite derivation process requires the elaboration of mapping rules, however this should be fast for a designer after some training. Generating detailed LOTOS behavior specifications takes much more work. Still, the whole process can be used at the design stage because of the high level of abstraction. The functional tests thus obtained can be valuable because they can be reused at various implementation stages, after suitable translation.

Our method distinguishes itself from others in two respects. First of all, it attacks the feature interaction problem with a logical analysis of abstract requirements. Because of the abstraction, not all feature interactions are covered.

However those that will be exposed will be exposed efficiently, without having to resort to the costly state exploration analysis that is the main tool of several other methods. The abstract nature of the method also makes it applicable well beyond the examples of this paper. The second original aspect of our method relates to the testing aspect. The test scenarios generated can be used to test for interactions at several phases of the software life-cycle.

T. Ohta and T. Yoneda developed a similar method [28, 29] for feature interaction detection. However, their method is not based on logical analysis and does not permit derivation of test cases. A detailed comparison with our technique can be found in [23].

Concerning scalability, we should note that real-life telephony systems include hundreds of features. However most of these are simple and clearly completely disjoint and do not need to be analyzed. Our industrial case study has shown, to the satisfaction of the industrial partner, that our method is applicable to real-life systems. A patent application has been submitted by the company.

In [9, 30], the method was extended, in two different ways, to detect interactions in user-defined features for Internet telephony. This extension is very interesting in practice, because users cannot be expected to be good designers of features and will easily produce interactions.

Many improvements are still possible on our method. We are extending our research for the elaboration of new rules in order to be able to identify new incoherencies such as incoherencies involving more than two features. Another consideration is to link our rules to the OPI (Obligation, Permission, Interdiction) [31] model. Refining rules based on this model will, for instance, allow to make a distinction between *Obliged*, *Permitted* and *Forbidden* results of a feature.

Acknowledgements This work was completed when all the authors were at the University of Ottawa. It was funded in part by Communications and Information Technology of Ontario, Mitel Corporation, and the Natural Sciences and Engineering Research Council of Canada.

References

1. Zave, P.: Architectural solutions to feature-interaction problems in telecommunications. In: Calder, M., Magill, E. (eds.) Feature Interactions in Telecommunications and software systems VI, pp. 10–22. IOS Press (1998)
2. Kimbler, K., Bouma, L.G.: Feature Interactions in Telecommunications and software systems VI. IOS Press (1998)
3. Calder, M., Magill, E.H.: Feature Interactions in Telecommunications and software systems VI. IOS Press (2000)
4. Amyot, D., Logrippo, L.: (Eds.). Feature Interactions in telecommunications and software systems VII. IOS Press (2003)
5. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: A critical review and considered forecast. In Computer Networks, vol 41 (1) 115–141. Elsevier Science (2003)
6. Reiff-Marganiec, S., Turner, K.J.: Feature interactions in policies. Computer Networks **45**(5), 569–584 (2004)
7. Dini, P., Clemm, A., Gray, T., Lin, F.J., Logrippo, L., Reiff-Marganiec, S.: Policy-enabled mechanisms for feature interactions: Reality, expectations, challenges. Computer Networks **45**(5), 585–603 (2004)
8. Cameron, E.J., Griffeth, N., Lin, Y., Nilson, M.E., Schnure, W.K., Velthuijsen, H.: A feature interaction benchmark for in and beyond. In Workshop on Feature Interactions in telecommunications Systems, pp. 1–23. IEEE Communications 31 (1993) 64–69 (revised and reprinted In: Bouma, L.G., Velthuijsen, H. (eds.), IOS Press, Amsterdam) (1994)
9. Amyot, D., Gray, T., Liscano, R., Logrippo, L., Sincennes, J.: Interactive conflict detection and resolution for personalized features. To appear in the Journal of Communications and Networks (2004)
10. Kimbler, K.: Addressing the interaction problem at the enterprise level. In Dini, P., Boutaba, R., and Logrippo, L. (eds.) Feature Interactions in Telecommunications Networks IV(FIW'97), pp. 13–22 (1997)
11. Metzger, A., Webel, C.: Feature interaction detection in building control systems by means of a formal product model. In: Amyot, D., Logrippo, L. (eds.) Feature Interactions in Telecommunications and software systems VII, pp. 105–121. IOS Press, (2003)
12. Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., Mankovski, S.: Feature-interaction visualization and resolution in an agent environment. In Kimbler, K. and Bouma, L.G. (eds.) Feature Interactions in Telecommunications and software systems V, pp. 135–149. IOS Press (1998)
13. Marples, D., Magill, E.H.: The use of rollback to prevent incorrect operation of features in intelligent network based systems. In: Kimbler, K. Bouma, L.G. (eds.) Feature Interactions in Telecommunications and software systems V, pp. 115–134. IOS Press (1998)
14. Jia, Y., Atlee, J.M.: Run-time management of feature interactions. In ICSE Workshop on Component-Based Software Engineering (CBSE6) (2003)
15. Black, U.: The Intelligent Network: Customizing Telecommunication Networks and Services. Prentice Hall (1998)
16. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. Internet Engineering Task Force (IETF), June (2002) RFC-3261
17. Lennox, J., Schulzrinne, H.: Call Processing Language Framework and Requirements. Internet Engineering Task Force (IETF), May 2000. RFC-2824
18. Bolognesi, T., Briskma, E.: Introduction to the ISO specification language LOTOS. In Computer Networks and ISDN Systems 14 (1987), pp. 25–59. Elsevier (1989)
19. Logrippo, L., Faci, M., Haj-Hussein, M.: An Introduction to LOTOS: Learning by Examples. Computer Networks and ISDN Systems **23**(5), 325–342, (1992). Errata in **25**(1), 99–100 (1992)
20. Turner, K.J.: Using Formal Description Techniques. J. Wiley & sons Ltd. (1993)
21. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
22. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
23. Gorse, N.: The feature interaction problem: Automatic filtering of incoherences & generation of validation test suites at the design stage. Master's thesis, University of Ottawa, Ottawa (2001)
24. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer, 4th edition (1994)
25. Wielemaker, J.: SWI Prolog reference manual. Dept. of Social Science Informatics (SWI) (2000)
26. Pavón, S., Larrabeiti, D., Rabay, G.: LOtos LAboratory user manual (version 3r6). Universidad Politécnica de Madrid (1995)
27. Amyot, D., Charfi, L., Gorse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., Ware, T.: Feature description and feature interaction analysis with Use Case Maps and LOTOS In: Calder, M., Magill, E. (eds.) Feature Interactions in Telecommunications and software systems VI, pp. 274–289. IOS Press (2000)

28. Yoneda, T., Ohta, T.: A formal approach for definitions and detection of feature interactions. In: Kimbler, K., Bouma, L.G. (eds.) Feature Interactions in Telecommunications and software systems V, pp. 202–216. IOS Press (1998)
29. Yoneda, T., Ohta, T.: Automatic elicitation of knowledge for detecting feature interactions in telecommunications services. In IEICE Transactions on Information and Systems, pp. 640–647. IEICE, Japan (2000)
30. Xu, Y., Logrippo, L., Sincennes, J.: Detecting feature interaction in CPL. To appear in the Journal of Network and Computer Applications (2004)
31. Barbuceanu, M., Gray, T., Mankowski, S.: How to make your agents fulfil their obligations. In Proceedings of PAAM-98 (1998)

Nicolas Gorse received a Master of Computer Science from the University of Ottawa, School of Information Technology and Engineering in 2001.
He is currently a Ph.D. candidate in the Département d'Informatique et Recherche Opérationnelle of the Université de Montréal. His research interests relate to formal methods and their application in the design and verification of complex electronic systems at high levels of abstraction.

Luigi Logrippo received a degree in law from the University of Rome (Italy) in 1961, and in the same year he started a career in computing. He worked for several computer companies and in 1969 he obtained a Master of Computer Science from the University of Manitoba, followed by a Ph.D. of Computer Science from the University of Waterloo in 1974.
He was with the University of Ottawa for 29 years, where he was Chair of the Computer Science Department for 7 years. In 2002 he moved to the Université du Québec en Outaouais, Département d'Informatique et Ingénierie, while remaining associated with the University of Ottawa as an Adjunct Professor.
His interest area is formal and logic-based methods and their applications in the design of communications systems. For a number of years he worked on the development of tools and methods for the language LOTOS. Current research deals with the formal analysis of advanced communications services made possible by internet telephony, of the policies that govern them, and of their interactions, in application areas such as presence features and e-commerce contracts.

Jacques Sincennes is a research programmer/systems analyst at the University of Ottawa, School of Information Technology and Engineering. He has held this position for the past 17 years. He is coauthor of a number of papers and a patent application.