Journal of King Saud University – Computer and Information Sciences (2013) 25, 99–115

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

ORIGINAL ARTICLE

A high abstraction level approach for detecting feature interactions between telecommunication services

Zohair Chentouf [a,*], Ahmed Khoumsi [b,1]

[a] Department of Computer Science, College of Computer and Information Science, King Saud University, Saudi Arabia
[b] Department of Electrical and Computer Engineering, University of Sherbrooke, Canada

Received 21 January 2012; revised 5 May 2012; accepted 1 October 2012
Available online 8 October 2012

KEYWORDS
Telecommunication services;
Feature interaction detection;
Service modeling;
Service management

Abstract When several telecommunication services are running at the same time, undesirable behaviors may arise, which are commonly called feature interactions. Several methods have been developed for detecting and resolving feature interactions. However, most of these methods are based on detailed models of services, which make them suffer from state space explosion. Moreover, different telecommunication operators cannot cooperate to manage feature interactions by exchanging detailed service models because this violates the confidentiality principle. Our work is a part of the few attempts to develop feature interaction detection methods targeting to avoid or reduce significantly state space explosion. In order to reach this objective, we first develop a so called Cause–Restrict language to model subscribers of telecommunication services at a very high abstraction level. A Cause–Restrict model of a subscriber provides information such as: what is the cause of what, and what restricts (or forbids) what, and specifies coarsely the frequency of each operation "cause" or "restrict" by "always" or "sometimes". Then, we develop a method that detects feature interactions between telecommunication services modeled in the Cause–Restrict language. We demonstrate the applicability of our approach by modeling several services and detecting several feature interactions between them. New feature interactions have been detected by our approach.

1. Introduction

With the great development of telecommunication systems, the number of services available to the users is continuously increasing for several years. However, undesirable behaviors arise when several services $S_1, \ldots, S_n$ are run together. Those undesirable behaviors are commonly called *feature interactions* (FIs). We say that there is an FI between $S_1, \ldots, S_n$ or that those services interact with each other. To be clear, let us give an example of services Originating Call Screening (OCS) and Call Forward Unconditional (CFU) and an FI OCS–CFU that occurs between them: A subscriber of OCS can put phone numbers in a list L_{OCS} so that every outgoing call from A

* Corresponding author.
E-mail addresses: zchentouf@ksu.edu.sa (Z. Chentouf), Ahmed.Khoumsi@Usherbrooke.ca (A. Khoumsi).
[1] On sabbatical leave at the College of Computer and Information Science, King Saud University when the study was done.
Peer review under responsibility of King Saud University.

toward a number in L_{OCS} is automatically blocked. A subscriber of CFU can program an automatic forward to a specific number so that all his incoming calls are automatically forwarded to that specific number. Consider A and B who are subscribers of OCS and CFU respectively, and a third user C. Assume that A has put C's number in his list L_{OCS} with the idea that a call initiated by A must not be established between A and C. Assume that B has programmed an automatic unconditional forward of his incoming calls towards C. OCS prevents that A calls directly C (who is in L_{OCS}), but the FI comes from the fact that A can join C by calling B who forwards the call to C although C is in L_{OCS}.

Once the services are analyzed together, some FIs may seem improbable with the impression that experienced designers of services should not make some "mistakes" that are causes of FIs. But when studying whether FIs arise between services, a fundamental assumption is that each service has been designed independently of the other ones. Not making this assumption is unrealistic in the general case, because during the design of a service S, it is practically impossible to consider all the existing services that may run together with S. Moreover, it is fundamentally impossible to consider the non-existing services that will be designed in the future.

1.1. Existing approaches to FI detection and resolution

FIs have been studied substantially in telecommunication systems (Bouma and Velthuijsen, 1994; Cheng and Ohta, 1995; Dini et al., 1997; Kimbler and Bouma, 1998; Calder and Magill, 2000; Amyot and Logrippo, 2003; Reiff-Marganiec and Ryan, 2005; Du Bousquet and Richier, 2007; Nakamura and Reiff-Marganiec, 2009), and more recently in other kinds of systems, such as web services (Weiss et al., 2007). We consider here uniquely telecommunication services, thus the term *service* will mean *telecommunication service*.

A solution to the FI problem is often operated in two phases. First, FIs are detected between services. Then, a FI resolution mechanism is executed to solve the detected interactions. Most of the FI detection research reported in the literature uses formal methods. Services are modeled using a formal language. Then, formal techniques are used in order to detect possible interactions. The commonly used formal techniques are temporal logic (Blom et al., 1995), theorem proving (Gammelgaard and Kristensen, 1994), Petri nets (Nakamura et al., 1997), extended finite state automata (SDL language, for example) (Gibson and Mery, 1997), and process algebra (LOTOS language, for example) (Amyot et al., 2000). Informal methods are also used to detect FIs. For example, Charnois (1997) uses natural language processing to identify interactions between service logic requirements modeled as textual descriptions. FI resolution uses two main methods: restriction and negotiation. Restriction consists in specifying a precedence or exclusion rule to apply in order to avoid a given FI. Precedence means to run one service before another and exclusion means to exclude one of the interacting services. For example, Cherkaoui and Khoumsi (2002) proposed a solution based on software agents which apply restriction rules. Other examples of restriction can be found in (Khoumsi, 1997; Blom et al., 1994; Tsang and Magill, 1997). The negotiation method to solve FIs usually uses software agents capable of communication and negotiation. Kolberg et al. (2002) designed negotiating agents which try to satisfy the preferences of end users and network operators. Amer et al. (2000) proposed an architecture that contains negotiating agents which represent end users and network devices. The work of Griffeth and Velthuijsen (1994) presents negotiating agents which represent end users' preferences.

When detection and resolution are performed at design time, they are qualified as off-line. They are qualified as on-line if they are performed at runtime. We may also have hybrid approaches where both off-line and on-line methods are performed (Calder et al., 2007).

Off-line approaches are generally based on formal methods and necessitate a great amount of information. For example, methods using model-checking techniques require going through a state space. The latter increases significantly with the complexity and number of services. Hence, those methods suffer from the problem of state space explosion. On-line approaches avoid the state space explosion by considering a state only when it is reached, instead of considering all possible states before they are reached. However, on-line methods have hard timing constraints since they are executed while services are running.

To provide a solution to the aforementioned state space explosion, some authors developed pragmatic off-line methods where services are modeled at a high abstraction level. For example, Kolberg and Magill (2007) designed a solution in which every service is abstracted by a triggering party, and origin and destination parties. As another example, Chentouf et al. (2004) abstracted services by some processing points that correspond to the main steps in a phone call, such as *OffHook, Dial, Wait, Response Received*, and *Speak*.

1.2. Our approach to FI detection

We adopt the same approach as (Chentouf et al., 2004; Kolberg and Magill, 2007) but with a more ambitious objective by going further in the abstraction level of services. The aim is to reduce the state space and avoid revealing details on service design. In order to reach this objective, we first develop a so called Cause–Restrict language (or more briefly CR-language) to model subscribers of services at a very high abstraction level. A CR-model (or CR-description) of a subscriber provides information such as: what is the cause of what, and what restricts what, and specifies coarsely the frequency of each operation "cause" or "restrict" by "always" or "sometimes". Then, we develop a method that detects FIs between CR-models of services. We demonstrate the applicability of our approach by modeling several concrete services and detecting several FIs between them. Our approach has permitted us to detect known FIs and, more interestingly, new FIs which we did not find in the literature.

1.3. Structure of the paper

The structure of this paper is as follows. Section 2 presents the CR-language which is used to model interfaces and behaviors of subscribers of services. In Section 3, we develop a Cause–Restrict-based method for detecting FIs. Section 4 demonstrates our FI detection method in several concrete examples. Section 5 compares the proposed approach with related works. Finally, we conclude in Section 6 by recapitulating our results and proposing some future work.

2. Cause–Restrict language to model interfaces and behaviors

2.1. Outline and Objective of the Cause–Restrict language

Some FI detection methods, like (Chentouf et al., 2004; Kolberg and Magill, 2007), reduced state space explosion by modeling services at a high abstraction level. In the present paper, we target a more ambitious objective by going further in the abstraction level. Another objective is to avoid revealing details on service design whenever different operators have to exchange their service models. In such a case, operators want to jointly manage feature interactions that involve services which are deployed in their networks. For that purpose, we develop a so-called cause–restrict language to model users of services at two levels:

– First level: *interface model*The interfaces of calls and users are modeled as empty objects, that is, by attributes and empty methods, i.e., each method is defined by just a signature without a body.
– Second level: *behavior model*The behaviors of methods are modeled at a high abstraction level by the so-called Cause–Restrict relations (CR-relations). The principle consists in specifying "who causes what" and "who restricts what". CR-relations also specify coarsely the "frequency" of the operations, by qualifying each "cause" and "restrict" by "always" or "sometimes". Such an omission of details is motivated by the desire to highly abstract models. A set of CR-relations modeling the behavior of a user is called its behavior model, or more precisely "CR-behavior model" to emphasize the use of CR-relations.

The composition of the interface and behavior models of a user is called its CR-model. The CR-model describes service users *logically*, in the sense that it specifies how a service behaves. The CR-model does not necessarily correspond to the service implementation. The CR-model is targeted uniquely to be manipulated by our proposed FI detection method. Our method detects FIs by manipulating CR-descriptions of the users of services that are run together. Hence, our FI detection method uses uniquely information available in CR-models. In this Section 2, we present the two levels of CR-modeling, which consist in modeling the interfaces and behaviors of users of services, respectively.

The interface modeling is inspired from object oriented programming (OOP), which manipulates the notions of classes and objects. Class names are in **bold** with the first letter non-capitalized, while object names are in *Italic* with the first letter capitalized. For simplicity, we will present minimal versions of classes modeling calls, basic users and subscribers of services. We will see that these minimal versions are sufficient for detecting several FIs.

Instead of specifying in detail the behavior of each method of a class, the behavior model consists in specifying expressions "U R V", where R is a relation and U and V may be a method invocation or a boolean expression. For any U (or V), we say "we have U" or "U is true" to mean that the corresponding method is invoked or the corresponding condition is satisfied. We have two relations, "cause" and "restrict", which are tuned by ! and ? in the following way:

– U cause! V means that when we have U, we will certainly have V;
– U cause? V means that when we have U, we will sometimes have V;
– U restrict! V means that when we have U, V is forbidden;
– U restrict? V means that when we have U, V is sometimes forbidden.

Each "U R V" is called Cause–Restrict relation, or more briefly CR-relation. We believe that if the designer makes an effort to write accurate CR-relations, it is possible to minimize the number of CR-relations with "cause?" and "restrict?". Indeed, in Sections 2.2,2.3,2.4 and Appendix A we present examples of interface models and CR-behavior models of basic users and subscribers of several services. We use "cause!" and "restrict!" only.

In the sequel, we will use the following terms with a generic meaning:

– A, B and C denote users
– *anyUser* denotes any user; for example, *anyUser* can be replaced by A, B or C.
– *anyCom* (*X*) denotes any method that performs a communication (using any media) with a user X.

These minimal versions will be sufficient to demonstrate our FI detection method in several examples (in Section 4).

*2.2. Classes **call** and **user***

The fundamental classes are **call** and **user**. Class **call** models the interface of an established call, and class **user** models the interface of a "basic" user, i.e., a user who subscribes to no specific service. We also define classes **calls** and **users** that model a set of established calls and a set of users, respectively. To give an idea of how the interfaces of a call and a user are modeled, we present below minimal versions of the classes **call** and **user**.

```
call:
// Attributes
user Caller               // Caller in the current call.
user Callee               // Callee in the current call.
user Initiator            // Initiator of the current call,
                          // the user who is the original cause of the
                          // call.
users Participants        // Participants in the current call
// Methods
void accept(user)         // Accept a user that joins the current call.
void end()                // Terminate the current call.
```

In a **call** object modeling a basic call, i.e., where no service is involved, the attributes *Initiator* and *Caller* are equal, and the attribute *Participants* consists uniquely of *Caller* and *Callee*. But this property is not guaranteed when services are involved.

user :	
// Attributes	
int *number*	// Phone number of the current user
boolean *busy*	// True when the current user is busy, i.e., // he cannot receive a call.
boolean *idle*	// True when the current user is not engaged // in a call, i.e., *Calls* = ∅.
boolean *noAnswer*	// True when the current user is busy and does // not answer.
calls *Calls*	// Set of calls where the current user is engaged
users *Connected*	// Set of users connected to the current user // through a call
// Methods	
call *call*(**user**)	// Initiates a call to a user and returns the // established call. // It returns *null* if no call is established.
call *acceptCall*(**user**)	// Accepts a call coming from a user and // returns the established call. // It returns *null* if no call is established.
void *busy*()	// Reaction to an incoming call when the // current user is busy.
void *noAnswer*()	// Reaction to an incoming call when the // current user does not answer.
void *serverFailure*()	// Reaction to an incoming call when // the server cannot process it.
void *endCall*(**call**)	// Terminates a call.
info *infoCaller*()	// The current user receives information // on the caller, // e.g., his phone number.

The interface of each user is modeled as a **user** object. Its attributes correspond to information on the user, like *number, busy, Calls* and *Connected*, which are explained above (as comments) in the **user** class definition. The methods correspond to functionalities, like *call*(), *acceptCall*(), *busy*(), *noAnswer*(), *serverFailure( )*, *endcall*() and *infoCaller*(), which are explained above in the **user** class definition.

2.3. Basic behavior model

The basic behavior model consists of CR-relations specifying how two basic users A and B (i.e., they subscribe to no specific service) behave in a call. Here are some CR-relations of such a basic behavior model (we have also to repeat all these CR-relations by switching between A and B):

1a: $Call = A.call(B)$	cause!	$A = Call.Initiator$
2a: $A = Call.Initiator$	cause!	$A \in Call.Participants$
3a: $Call = A.call(B)$	cause!	$B \in Call.\ Participants \setminus \{Call.Initiator\}$
4a: $(A.idle = \text{true})$	restrict!	$(A.busy = \text{true})$
5a: $(A.idle = \text{false})$	cause!	$(A.busy = \text{true})$
6a: $(A.noAnswer = \text{true})$	restrict!	$(A.busy = \text{true})$

The CR-relations obtained by switching A and B are numbered from 1b to 6b.

In the 1st and 3rd CR-relations, "$A.call(B)$" means that A calls B, and the fact to write "$Call=$" means that a reference to a call is returned, hence the call initiated by A toward B is established. And the 2nd CR-relation means that the initiator of a call is certainly a participant of that call ($Call.Initiator \in Call.Participants$). The 3rd CR-relation means that if A calls successfully B, B is certainly a participant, but not the initiator, of the call. The 4th and 5th CR-relations mean that idle and busy are exclusive status. The 6th CR-relation points out the semantics of "$A.noAnswer$ = true", which is that A is not busy and does not answer.

The attribute *idle* may seem redundant with *busy*, because *idle* = true if and only if *busy* = false, that is, *idle* is the negation of *busy*. We have defined it because the negation relation between *idle* and *busy* does not hold with some specific services. For example, the subscriber of Multiple Lines (ML) service (Appendix A.7) is busy when *all* his lines are busy, and he is idle when *all* his lines are idle. Hence, when some (but not all) lines are busy, the subscriber of ML is neither busy nor idle.

2.4. Subscriber of a service

The interface of a subscriber of a service S is modeled by a class named **userS** that inherits from the class **user** by adding attributes and modifying and/or adding functionalities. More precisely:

- A class **userS** may add one or more new attributes that are not present in the parent class **user**. A new attribute represents a status related to the service. For example, if a service S can be enabled/disabled by his subscriber, we can use a boolean attribute that specifies whether S is enabled or disabled. The latter type of attribute is qualified as *generic* because it is defined in a generic way. We can also have *specific* attributes that are defined only for a specific service.
- A class **userS** necessarily behaves differently than the parent class **user**, for example by handling new attribute(s). Such a different behavior is possible only by modifying functionalities of the parent class **user** and/or by adding new functionalities that are absent in **user**. In the interface modeling, adding functionalities is modeled by defining new methods, while modifying functionalities is modeled by overriding methods of **user**. Recall that in the interface modeling, methods are defined just by their signature. Their behaviors are defined by properties in the behavior model (second level of the CR-model).

The behavior model of a subscriber of a service S is defined by adding and/or removing properties of the basic behavior model.

To give an idea on interface and behavior models, we present in the following Sections 2.4.1,2.4.2,2.4.3,2.4.4,2.4.5,2.4.6 several service subscribers. Other services subscribers are presented in Appendix A.

2.4.1. Subscriber of Call Forward Unconditional (CFU)

A subscriber of CFU can program an automatic forward to a user so that all his incoming calls are automatically forwarded to the specified user.

```
Interface model: subscriber of CFU
userCFU extends user
// Attributes
user forward                  // Specific attribute indicating
                              // the user to whom incoming calls
                              // are forwarded.
// Methods
call acceptCall (user)        // Overrides the method
                              // acceptCall() of the class user
```

The method *acceptCall*() of **user** is overridden because acceptance of an incoming call by CFU consists in forwarding the call.

CR-behavior model: subscriber B of CFU

In the basic behavior model, the CR-relation 3a is removed and the following CR-relations are added:

$(B.forward = C) \wedge A.call(B)$ cause! $B.call(C)$

$(B.forward = C) \wedge (Call = A.call(B))$ cause! $C \in Call.Participants$

In these CR-relations, "*B.forward = C*" means that B has programmed an automatic forward toward C. Hence, the 1st CR-relation means that if A calls B who has programmed a forward to C, then B will automatically call C. And the 2nd CR-relation means that if A calls successfully B who has programmed a forward to C, then C is certainly a participant of the call.

2.4.2. Subscriber of Terminating Call Screening (TCS)

A subscriber of TCS registers users in a list L_{TCS} so that every incoming call from a user registered in L_{TCS} is automatically blocked.

```
Interface model: subscriber of TCS
userTCS extends user
// Attributes
users ListTcs                 // Specific attribute
                              // corresponding to
                              // L_TCS of the current subscriber of
                              // TCS.
// Methods
call acceptCall(user)         // Overrides the method accept
                              // Call() of the class user.
```

The method *acceptCall*() of **user** is overridden because incoming calls from users registered in L_{TCS} must not be established.

CR-behavior model: subscriber B of TCS

The following CR-relations are added to the basic behavior model:

$(A \in B.ListTcs) \wedge A.call(B)$restrict! $A.anyCom(B)$

$(A \in B.ListTcs) \wedge A.call(B)$restrict! $B.anyCom(A)$

$(A \in B.ListTcs) \wedge (B \in Call.Participants \setminus \{Call.Initiator\})$ restrict! $A \in Call.Participants$

$(A \in B.ListTcs) \wedge (A = Call.Initiator)$ restrict! $B \in Call.Participants$

The 1st and 2nd CR-relations mean that if A is in L_{TCS} of B and A calls B, then A cannot be in communication with B. For that purpose, if A is in L_{TCS} of B and A calls B, we forbid that any communication method of A be called with B as argument and that any communication method of B be called with A as argument. This is a guarantee that no communication initiated by A can be established between A and B. The 3nd CR-relation means that if A is in L_{TCS} of B and B participates in a call he has not initiated, then A cannot be a participant of that call. The 4th CR-relation means that if A initiates a call and is in L_{TCS}, then B cannot be a participant of that call.

2.4.3. Subscriber of Automatic Recall (AR)

A subscriber B of AR can enable AR so that if B is called from any user A while he is busy, then a call is automatically generated from B to A as soon as B is idle again.

```
Interface model: subscriber of AR
userAR extends user
// Attributes
boolean ar                    // Generic attribute indicating
                              // whether AR is enabled.
// Methods
void busy()                   // Overrides the method
                              // busy() of the class user.
```

The method *busy*() of **user** is overridden because AR modifies the reaction to incoming calls when B is busy.

CR-behavior model: subscriber B of AR

The following CR-relation is added to the basic behavior model:

$(B.ar = \text{true}) \wedge Call = A.call(B) \wedge (B.busy = \text{true})$ cause!
$B.call(A) \wedge (B \in Call.Participants \setminus \{Call.Initiator\}) \wedge (A = Call.Initiator)$

The above CR-relation means that if AR is enabled and A calls successfully B who is busy, then B will call A in order to establish the call initiated by A.

2.4.4. Subscriber of Call Waiting (CW)

If a subscriber B of CW is called from a user A while B is in communication with a user C, then A is put on hold. Then, B can put C on hold and connect to A. B can switch between A and C.

```
Interface model: subscriber of CW
userCW extends user
// Attributes: no
// new attribute is
// defined
// Methods
void busy()                   // Overrides the method busy() of
                              // the class user.
call hold(user)               // New method: it accepts a call
                              // coming from a user but puts him
                              // on hold; it
                              // returns the established call; It
                              // returns null if no call is
                              // established.
void hold(call, user)         // New method: it puts on hold a
                              // user participating in a call.
```

The method *busy*() of **user** is overridden because CW modifies the reaction to incoming calls when B is busy. Two new methods *hold*() are added.

CR-behavior model: subscriber B of CW

The following CR-relation is added to the basic behavior model:

$$A.call(B) \wedge (B.busy = \text{true})\ \text{cause!} B.hold(A)$$

The above CR-relation means that if A calls B who is busy, then B will put A on hold.

2.4.5. Subscriber of Unified Messaging (UM)

If a user A calls a subscriber B of UM who is busy or does not answer, then A is forwarded to a voicemail server to leave a voice message to B. What has just been said corresponds to the service named Voicemail (VM). We obtain UM from VM by requiring that the voice message left by A is sent by email to B.

Interface model: subscriber of UM	
userUM extends	
user	
// Attributes	
boolean *um*	//Generic attribute indicating
	// whether UM is enabled.
// Methods	
call	// Overrides the method
acceptCall(**user**)	// *acceptCall*() of the class **user**.
void	// *unifiedMessaging*(**user, voice**)
// New method:	
// it receives by	
// email a voice	
// message from a	
// user.	

The method *acceptCall*() of **user** is overridden because UM modifies the reaction to incoming calls. A new methods *unifiedMessaging*() is added.

CR-behavior model: subscriber B of UM

The following CR-relation is added to the basic behavior model:

$$(B.um = true) \wedge A.call(B) \wedge (B.busy = true \vee B.noAnswer = \text{true})\ \text{cause!}$$
$$B.unifiedMessaging(A,\ \text{VoiceMsg})$$

The above CR-relation means that if UM is enabled and A calls B who is busy or does not answer, then B will receive from A an email containing a recorded voice message.

2.4.6. Subscriber of Follow-Me (FM)

A subscriber of FM can specify a list of numbers where to join him in a given order. That is, if the first number is not busy and does not answer, then the second number is tried and so on, until one of the numbers answers or none of the numbers answers. The first number in the list is considered as the subscriber of FM, and the other numbers constitute an ordered list L_{FM}. Conceptually, this is equivalent to defining a list of users to join in a given order until one of them answers or all the list is tried without answer. The latter behavior is called Hunt Group or Group-Calling service. Here, we model the FM service only because Group-Calling is conceptually equivalent to it.

Interface model: subscriber of FM	
userFM extends user	
// Attributes	
users *ListFm*	// Specific attribute corresponding
	// to L_{FM} of the subscriber of FM
user *FirstAnswer*	// Specific attribute corresponding
	// to the first number in L_{FM}
	// (if any) who
	// answers while the preceding
	// numbers in the list are not busy
	// and do not
	// answer.
// Methods	
void *noAnswer*()	// Overrides the method
	//*noAnswer*() of the class **user**.

The method *noAnswer()* of **user** is overridden because a new behavior is triggered by "No answer" in FM: trying the next number.

CR-behavior model: subscriber B of FM

In the basic behavior model, the CR-relation 3a ($Call = A.call(B)$ cause! $B \in Call.Participants \setminus \{Call.Initiator\}$) is removed and the following CR-relations are added:

$$(Call = A.call(B)) \wedge (B.noAnswer = \text{true})\ \text{cause!}(B.FirstAnswer \in Call.Participants)$$
$$(Call = A.call(B)) \wedge (B.noAnswer = \text{true})\ \text{restrict!}\ (B \in Call.Participants)$$

The above CR-relations mean that if A calls B who is not busy and does not answer, then the first user in L_{FM} who answers (if any) becomes a participant of the call while B is not a participant of the call.

2.5. Subscriber of several services

In Section 2.4, we have shown how to specify the interface and behavior of a subscriber of a service S: the interface is constructed by using inheritance from the class **user**. This approach can be generalized to specify the interface of a subscriber of several services as follows: a subscriber of several services $S_1,\ldots,S_n$ can be considered as several subscribers $\text{Subs}_1,\ldots,\text{Subs}_n$, where each Subs_i has a single service S_i. Hence, the interface of each Subs_i is modeled as an object of a class **userSi** inheriting from the class **user**.

Note that in the interface specification, we cannot specify the difference between a method of **user** and a method with the same name that overrides it in **userS**. For example: *acceptCall*() which are used in **user**, **userCFU**, **userTCS**; and *busy*() which are used in **user**, **userAR**, **userCW**. The difference between the methods with the same name can be specified in the behavior model.

2.6. Less accurate CR-Relations: cause? restrict?

All the CR-relations in Subsections 2.3 and 2.4 use "cause!" and "restrict!", while "cause?" and "restrict?" are never used. This is because we have made an effort to write CR-relations as accurate as possible. But if information is removed from a CR-relation, we may have to replace "!" by "?". Let us show this by using a few examples:

Subscriber of CFU

If, in Section 2.4.1, we remove "*B.forward* = *C*", we have to replace "cause!" by "cause?":

$A.call(B)$cause? $(B.call(C))$

$(Call = A.call(B))$ cause?$C \in Call.Participants$

Intuitively, in Section 2.4.1, we are more accurate by assuming that B has programmed a forward toward C, while here the assumption is removed.

Subscriber of TCS

If, in Section 2.4.2, we remove "$B \in Call.Participants \setminus \{Call.Initiator\}$" from the 3rd CR-relation, we have to replace "restrict!" by "restrict?":

$(A \in B.ListTcs)$ restrict?$A \in Call.Participants$

Intuitively, in the 3rd CR-relation of Section 2.4.2, we are more accurate by assuming that B participates in a given call without being the initiator of that call, while here the assumption is removed.

Subscriber of AR

If, in Section 2.4.3, we remove "*A.ar* = true" and/or "*B.busy* = true", we have to replace "cause!" by "cause?":

$$(Call = A.call(B)) \text{ cause? } B.call(A) \wedge (B \in Call.Participants \setminus \{Call.Initiator\}) \wedge (A = Call.Initiator)$$

Intuitively, in Section 2.4.3, we considered only the situation where AR is enabled and B is busy, while here the assumption is removed.

Subscriber of CW

If, in Section 2.4.4, we remove "*B.busy* = true", we have to replace "cause!" by "cause?":

$A.call(B)$ cause?$B.hold(A)$

Intuitively, in Section 2.4.4, we are more accurate by assuming that B is busy, while here the assumption is removed.

Subscriber of UM

If, in Section 2.4.5, we remove "$(B.um = \text{true}) \wedge (B.busy = \text{true} \vee B.noAnswer = \text{true})$", we have to replace "cause!" by "cause?":

$A.call(B)$ cause? $B.unifiedMessaging(B$, VoiceMsg).

Intuitively, in Section 2.4.5, we are more accurate by assuming that UM is enabled, while here the assumption is removed.

Subscriber of FM

If, in Section 2.4.6, we remove "*B.noAnswer* = true", we have to replace "cause!" by "cause?" and "restrict!" by "restrict?":

$(Call = A.call(B))$ cause? $(B.FirstAnswer \in Call.Participants)$

$(Call = A.call(B))$ restrict? $(B \in Call.Participants)$

Intuitively, in Section 2.4.6, we are more accurate by assuming that the subscriber of FM (i.e., the first number in the list) is busy or does not answer, while here the assumption is removed.

2.7. Some properties and rules of CR-relations

This section presents some rules of CR-relations. Two CR-relations are said incompatible if it is nonsense (or impossible) to have them together in the same CR-behavior model. Incompatible CR-relations are symptoms of FIs.

"U cause! V" and "U restrict! V" are incompatible,
"U cause! V" and "U restrict? V" are incompatible,
"U cause? V" and "U restrict! V" are incompatible.

Note that "U cause? V" and "U restrict? V" are not incompatible.

Before continuing, we need to define the notions of *weaker* and *stronger* CR-relations. A CR-relation M_1 is said to be stronger than a CR-relation M_2 if M_1 implies M_2; we can also say that M_2 is weaker than M_1.

Below are four rules that permit to derive a weaker CR-relation from an existing CR-relation; these rules are easily understandable from the fact that "cause!" and "restrict!" are stronger than "cause?" and "restrict?" respectively, and $X \wedge Y$ implies both X and Y:

R1: "U ∧ Z cause! V" = > "U cause? V",
R2: "U ∧ Z restrict! V" = > "U restrict? V",
R3: "U cause! V ∧ Z" = > "U cause! V".
R4: "U cause? V ∧ Z" = > "U cause? V".

If we take Z equal to true, R3 and R4 become trivial, while R1 and R2 become:

r1: "U cause! V" = > "U cause? V",
r2: "U restrict! V" = > "U restrict? V".

Note that R1 and R2 have been used to obtain the CR-relations of Section 2.6 from the CR-relations of Section 2.4.

To make the FI detection as efficient as possible, we should enrich the resulting CR-behavior model by using the rules below to derive a CR-relation from *two* existing CR-relations. The enrichment is motivated by the fact that CR-relations are used to detect FIs, hence the more we have CR-relations the more we can detect FIs. Another approach is to have a non-enriched specification and to apply the enrichment during FI detection. That is, the enrichment could be moved from "before FI detection" to "during FI detection". But this will make the FI detection more complex and enrichment will be executed several times (at each FI detection).

R5: "U cause! V" and "V ∧ Z cause! W" = > " U ∧ Z cause! W"
R6: "U cause? V" and "V ∧ Z cause! W" = > "U ∧ Z cause? W"
R7: "U cause! V" and " V ∧ Z restrict! W" = > "U ∧ Z restrict! W"
R8: "U cause? V" and " V ∧ Z restrict! W" = > "U ∧ Z restrict? W"

Below are other rules that permit to enrich the CR-behavior model:

R9: "U cause! V ∧ Z" and "V cause! W" = > " U cause! W ∧ Z"
R10: "U cause? V ∧ Z" and "V cause! W" = > " U cause? W ∧ Z"

If we take Z equal to true in R5–R8, we obtain the following rules r5–r8, while if we take Z equal to true in R9–R10, we obtain r5–r6.

r5: "U cause! V" and "V cause! W" = > "U cause! W"
r6: "U cause? V" and "V cause! W" = > "U cause? W"

r7: "U cause! V" and "V restrict! W" = > "U restrict! W"
r8: "U cause? V" and "V restrict! W" = > "U restrict? W"

Note that rules R5–R10 and r5–r8 are in the form "U R1 V" and "V $\wedge$ Z R2 W" = > "U $\wedge$ Z R W" where R2 equals to "cause!" or "restrict!", and R1 can be "cause!" or "cause?". This is because we cannot deduce a new CR-relation when R2 equals to "cause?" or "restrict?". Let us explain this in the following two examples:

- U causes certainly V (U cause! V) and V causes W only when it is not caused by U (V cause? W). This situation does not allow us to deduce that U causes W.
- U causes certainly V (U cause! V) and V forbids W only when it is not caused by U (V restrict? W). This situation does not allow us to deduce that U forbids W.

Other rules can be found, but the above ones are sufficient for a good comprehension of the CR-language and for our FI detection method.

2.8. Discussion on how to onstruct interface and Behavior Models

The interface and behavior models do not correspond necessarily to two consecutive steps. Actually, the two tasks are intimately related. We may start by determining intuitively fundamental properties and then we determine attributes and methods that permit to express those properties formally. The task sequence that should be used is the following:

1. We intuitively determine properties the non-respect of which is judged potential and problematic.
2. We determine attributes and methods (the latter by their signature) that are necessary to express formally the properties of Item 1.
3. We formally express the properties of Item 1.

Interface and behavior models are obtained at the terms of Items 2 and 3, respectively. Note that Item 1 precedes interface and behavior models.

3. FI detection method based on Cause–Restrict language

There exist many FI detection methods using detailed and complex specifications of services as inputs, and applying model-checking techniques to detect FIs *automatically*. Those methods present the advantage of having a high *power* of detection, but their main drawback is their *state space complexity*.

There exists no miraculous solution to this problem, we can reduce it, but by accepting a smaller power of detection and/or a less automatic detection process. This is the approach we have adopted. In order to reduce the state space complexity, we model subscribers of services in the CR-language. Actually, instead of detecting FIs with certitude, our method draws the attention on suspected FIs, which then need to be checked (automatically or manually). This is the price to pay to reduce very significantly the state space complexity.

In this Section, we propose an off-line cause–restrict-based method for detecting FIs between services. The approach is to have a CR-behavior model of each type of subscriber of service and to merge the CR-behavior models of the subscribers of services that are run together. FIs are detected by analyzing the resulting CR-behavior model.

3.1. Inputs: CR-behavior models

During the design of a service S_1, we have to check whether there exist FIs between S_1 and existing services $S_2,\ldots,S_n$ that may have to be run together with S_1. When we say that services $S_1,\ldots,S_n$ are run together, we mean that subscribers of those services are engaged in a same call session. The CR-behavior models of the subscribers of $S_1,\ldots,S_n$ are the inputs of the FI detection procedure. An approach is to require that the service provider that is the owner of any deployed service S provides the CR-behavior model of a subscriber of S; this CR-behavior model will be available for designers of new services.

3.2. Step 1: Merging and enriching the CR-behavior models

In order to detect FIs between $S_1,\ldots,S_n$ that may have to be run together, we merge the CR-behavior models of their subscribers. The merging consists in obtaining a single CR-behavior model by putting together the *n* CR-behavior models. After the merging, the resulting CR-behavior model is enriched in the following two ways:

(a) The rules R5–R10 and r5–r8 of Section 2.7 are applied maximally. That is, we synthesize all the new CR-relations that can be obtained from those rules. This can be done by using a fix-point method, which repeats the application of the rules until no new CR-relation is generated. The method converges because the number of possible CR-relations is finite. Note that this enrichment is automatable. Note that rules R1–R4 and r1–r2 are not used here because they permit to derive weaker CR-relations, which is not relevant for FI detection.
(b) New CR-relations can be added for stating relations between variables or methods of various services. For example, a relation between a variable of S_1 and a variable of S_2. This enrichment is generally non automatable and is realized by the designer in order to model relations he has identified.

3.3. Step 2: FI Detection

In the sequel, by "cause" we mean "cause!" or "cause?", and by "restrict" we mean "restrict!" or "restrict?". As already mentioned, our method does not target to indicate FIs with certitude, it rather draws the attention on suspected FIs, which then need to be checked (automatically or manually). FI detection consists in analyzing the whole CR-behavior model obtained in Step 1, and in generating an FI detection verdict. The analysis consists in checking the existence of the following FI patterns in the CR-behavior model obtained in Step 1. In the following, U and V are said to be compatible if and only if we can have them at the same time.

(a) **cause–loop:** There exists a series of CR-relations "$U_1 \wedge Y_1$ cause U_2", …,"$U_i \wedge Y_i$ cause U_{i+1}",…,"$U_n \wedge Y_n$ cause U_1" such that U_1 is a method invocation. This is a symptom of *loop* (or *cycle*) that may induce a blocking behavior. This is more understandable with the simple form where all Y_i equal to true, which gives "U_1 cause U_2",…,"U_n cause U_1". In fact, we obtain the simple form by applying rule R1 of Section 2.7 to the general form. Hence, instead of considering the general form of the cause–loop pattern, an alternative is to first apply rules R1 and then to consider uniquely the simple form.In the presence of a cause–loop symptom, we have to check if the cycle or blocking really occurs and if it is problematic. Cause–loop is illustrated in Section 4.1.1.

(b) **cause–restrict:** There exists a pair of CR-relations “U cause W” and “V restrict W” such that we have no certitude that U and V are incompatible. More precisely, either we have the certitude that U and V are compatible, or we are uncertain of their compatibility. This is a symptom of *conflict* (or *contradiction*), where an action or condition W may be at the same time caused and forbidden. A particular case is when U equals to V, that is, W may be at the same time caused and forbidden by the same U.In the presence of a cause–restrict symptom, we have to check if there really exists a situation where an action (or condition) is at the same time caused and forbidden. Cause–restrict is illustrated in Sections 4.2.1 and 4.2.2.Note that there is no symptom of FIs if we are certain that U and V are incompatible.

(c) **cause–cause:** There exists a pair of CR-relations “U $\wedge$ Y cause V” and “U $\wedge$ Z cause W”, such that we have no certitude that Y and Z are incompatible. More precisely, either we have the certitude that Y and Z are compatible, or we are uncertain of their compatibility. This may be a symptom of conflict or confusion. This is more understandable in the particular case where Y and Z equal to true, which gives “U cause V” and “U cause W”. Intuitively, the same U causes both V and W.In the presence of a cause–cause symptom, we have to check if V and W are, for example, incompatible or redundant. Section 4.3.1 contains an example of cause–cause corresponding to a conflict. Section 4.3.2 contains an example of cause–cause which corresponds to a conflict only under specific conditions.

Fundamentally, cause–restrict can be seen as a particular case of cause–cause, because “V restrict W” can be written “V cause neg(W)”, where neg(W) denotes the negation of W. Hence, the pattern “U cause W” and “V restrict W” can be written “U cause W” and “V cause neg(W)”. But we preferred to keep the two patterns so that the designer can select the one which is closer to the intuition, depending on the example.

4. Examples of FI detection

In this section, we apply the method described in Section 3 for the detection of several FIs between two services among the services presented in Section 2. Other examples are given in Appendix B. Each FI is named in the form S_1–S_2, where S_1 and S_2 are the two services that interact. Each FI S_1–S_2 is firstly presented intuitively by a context and a scenario:

– *Context*: we indicate the users involved in the FI; we also indicate the user who subscribes to each service S_1 or S_2 and specify how each S_1 and S_2 are programmed.
– *Scenario of FI*: We present an example of execution where the FI S_1–S_2 arises.

Then, we show how the FI is detected using the FI detection method of Section 3. Sometimes, we conclude by indicating other resembling FIs. The resemblance is meant in the way the FI is detected at the formal level.

4.1. Cause–loop FI

4.1.1. CFU–CFU

Context: Consider A and B who are subscribers of CFU. Assume that A (resp. B) has programmed an automatic forward of his incoming calls towards B (resp. A).

Scenario of FI: A calls B who calls A. We have a loop (or cycle) of actions.

Let us now show how this FI is detected by our FI detection method. By adapting the 1st CR-relation of Section 2.4.1, we obtain:

– For the subscriber B of CFU that has programmed a forward toward A:

$$\mathrm{M1}: (B.forward = A) \wedge\ A.call(B)\ \text{cause!}(B.call(A))$$

– For the subscriber A of CFU that has programmed a forward toward B:

$$\mathrm{M2}: \ (A.forward = B) \wedge\ B.call(A)\ \text{cause!}(A.call(B))$$

The pair of CR-relations (M1,M2) constitutes a cause–loop symptom in the form “$U_1 \wedge Y_1$ cause U_2”, “$U_2 \wedge Y_2$ cause U_1”, where U_1 = ”*A.call*(*B*)”, U_2 = “*B.call*(*A*)”, Y_1 = “*B.forward* = *A*” and Y_2 = “*A.forward* = *B*”. We have checked that the corresponding loop *A.call*(*B*)–*B.call*(*A*)–*A.call*(*B*) can occur (see the above *Context-Scenario*).

We obtain resembling FIs if we replace CFU by other versions of Call Forward, like Call Forward on Busy Line (CFBL), Call Forward on No Reply (CFNR), Call Forward on Time (CFT). We can hence define various combinations of FIs, like CFU–CFBL, CFBL–CFBL, CFNR–CFBL, etc.

4.2. Cause–restrict FI

4.2.1. AR–TCS

Context: Consider two users A and B, where B is subscriber to TCS and AR. Assume that B has put A in his list L_{TCS}, with the idea that a call involving B but not initiated by B has no effect on A (hence B is not authorized to be joined by A).

Scenario of FI: TCS prevents that B is directly joined by A. But consider that A calls B while B is busy. As soon as B becomes not busy, an automatic recall will connect B to A (and B is joined by A), although A is in L_{TCS}. Here, we say that B is joined by A (and not that B joins A), because A is the initiator of the call in the sense that A is the original cause of the call.

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of TCS that has put A in L_{TCS}, we have by using the 3rd CR-relation of Section 2.4.2:

$$\mathrm{M1}: (A \in B.ListTcs) \wedge (B \in Call.Participants \setminus \{Call.Initiator\})\text{restrict!}\ A \in Call.Participants$$

– For the subscriber B of AR that has programmed an automatic recall and receives a call from A while he is busy, we have by using CR-relation of Section 2.4.3:

$$\mathrm{M2}: (B.ar = \text{true}) \wedge (Call = A.call(B)) \wedge (B.busy = \text{true})\ \text{cause!}\ B.call(A) \wedge\ (B \in Call.Participants \setminus \{Call.Initiator\}) \wedge\ (A = Call.Initiator)$$

– For the user A that initiates a call, we have by using the 2nd CR-relation of Section 2.3:

$$\mathrm{M3}: A = Call.Initiator\ \text{cause!}\ A \in Call.Participants$$

By applying rule R9 To M2 and M3 with V = ” *A* = *Call.Initiator*”, W = ” *A*∈*Call.Participants*”, U = ”(*B.ar* = true) $\wedge$

$(Call = A.call(B)) \wedge (B.busy = \text{true})$", Z=" $B.call(A) \wedge (B \in Call.Participants \setminus \{Call.Initiator\})$", we obtain:

$$N1: (B.ar = \text{true}) \wedge (Call = A.call(B)) \wedge (B.busy = \text{true}) \text{ cause! } B.call(A) \wedge (B \in Call.Participants \setminus \{Call.Initiator\}) \wedge (A \in Call.Participants)$$

By applying rule R3–N1 with Z="$B.call(A) \wedge (B \in Call.Participants \setminus \{Call.Initiator\})$", we obtain:

$$N2: (B.ar = \text{true}) \wedge (Call = A.call(B)) \wedge (B.busy = \text{true}) \text{ cause! } (A \in Call.Participants)$$

The pair of CR-relations (M1,N2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where the participation of A in a call is at the same time implied by AR and forbidden by TCS (see the above *Context-Scenario*).

We obtain a resembling FI AR–OCS if we replace "B is subscriber of TCS" by "A is subscriber of OCS" (OCS is presented in Appendix A.1).

4.2.2. TCS–FM

Context: Consider B and C who are subscribers to FM and TCS respectively, and a third user A. Assume that C has put A in his list L_{TCS} with the idea that a call involving C but not initiated by C has no effect on A (hence C is not authorized to be joined by A). Assume that C is the first user in L_{FM} who answers.

Scenario of FI: TCS prevents that C is directly joined by A in a basic call. But if A calls B and B does not answer the call, C will be joined by A although A is in L_{TCS}.

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of FM who is called by A, we have by using the CR-relation of Section 2.4.6:

$$M1: (Call = A.call(B)) \wedge (B.noAnswer = \text{true}) \text{ cause!}(B.FirstAnswer \in Call.Participants)$$
$$M2: (Call = A.call(B)) \wedge (B.noAnswer = \text{true}) \text{ restrict! } (B \in Call.Participants)$$

– For the subscriber C of TCS that has put A in L_{TCS}, , we have by adapting the 4th CR-relation of Section 2.4.2:

$$M3: (A \in C.ListTcs) \wedge (A = Call.Initiator) \text{ restrict! } C \in Call.Participants$$

Since $C = B.FirstAnswer$ (see assumption in the above *Context*), the pair of CR-relations (M1,M3) constitutes a cause–restrict pattern, which is a symptom of conflict. We have checked that we can reach a situation where the participation of C in a call is at the same time implied by FM and forbidden by TCS (see the above *Context-Scenario*).

We obtain a resembling FI if we replace "C subscriber of TCS" by "A subscriber of OCS" (OCS is presented in Appendix A.1).

4.3. Cause–cause FI

4.3.1. AR–CW

Context: Consider two users A and B, where B is a subscriber to AR and CW.

Scenario of FI: If A calls a busy user B, A is put on hold by B (according to CW) while A is recalled back later by B (according to AR). But, it is nonsense to execute the two actions because they target the same objective, which is to put A and B in communication. So, which action should be executed?

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of AR that has programmed an automatic recall and receives a call from A while he is busy, we have by using the CR-relation of Section 2.4.3:

$$M1: (B.ar = \text{true}) \wedge (Call = A.call(B)) \wedge (B.busy = \text{true}) \text{ cause! } B.call(A) \wedge (B \in Call.Participants \setminus \{Call.Initiator\}) \wedge (A = Call.Initiator)$$

– For the subscriber B of CW that receives a call from A while he is busy and puts A on hold, we have by using the CR-relation of Section 2.4.4:

$$M2: A.call(B) \wedge (B.busy = \text{true}) \text{ cause!} B.hold(A)$$

The pair of CR-relations (M1,M2) constitutes a cause–cause pattern. We have checked that we can reach a situation where B calls A according to AR while B puts A on hold according to CW (see the above *Context-Scenario*).

4.3.2. CW–UM

Context: Consider two users A and B, where B is subscriber to CW and UM.

Scenario of FI: If A calls a busy B, A is put on hold by B (according to CW) while B receives a voice message by email from A (according to UM). Which of the two actions should be executed? Should we execute both actions?

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of CW that receives a call from A while he is busy and puts A on hold, we have by using the CR-relation of Section 2.4.4:

$$M1: A.call(B) \wedge (B.busy = \text{true}) \text{ cause!} B.hold(A)$$

– For the subscriber B of UM that receives a call from A and then sends him a voicemail message, we have by using the CR-relation of Section 2.4.5:

$$M2: (B.um = \text{true}) \wedge A.call(B) \wedge (B.busy = \text{true} \vee B.noAnswer = \text{true}) \text{ cause! } B.unifiedMessaging(A, \text{VoiceMsg})$$

The pair of CR-relations (M1,M2) constitutes a cause–cause FI symptom. We have checked that we can reach a situation where B puts A on hold according to CW while B receives a voice message by email according to UM (see the above *Context-Scenario*).

We obtain a resembling FI if we replace "B subscriber of UM" by "B is subscriber of VM" or "A subscriber of Email" (Email is presented in Appendix A.2).

4.4. Recapitulation

In addition to the FIs of Section 4 that involve the services of Section 2.4, we have also detected the FIs of Appendix B that involve also the services of Appendix A. Table 1 outlines the FIs we have detected between every pair of the thirteen services of Section 2.4 and Appendix A. The services presented in

Table 1 Detected FIs.

	CFU	OCS	TCS	AR	CW	UM	Eml	Feml	CC	FM	BLF	PIN	ML
CFU	cl												
OCS	cr	x											
TCS	cr	x	x										
AR	cc*	cr	cr	cl*									
CW	cc*	cr*	cr*	cc	x								
UM	cc*	cr	cr	cc	cc	x							
Email	cc*	cr	cr	cc	cc	x	x						
F-email	x	x	x	x	x	cr	cr	x					
CC	x	cr	cr	x	x	x	x	x	x				
FM	cc*	cr	cr	x	x	cc*	cc*	x	x	x			
BLF	x	x	x	x	x	X	x	x	x	x	x		
PIN	x	x	cr	cc*	x	X	x	x	x	x	cc*	x	
ML	x	x	x	cr*	cr*	cr*	x	x	x	x	cr	x	x

Section 2.4 are indicated in italic. The pattern of each detected FI is indicated by cl (cause–loop), cr (cause–restrict), and cc (cause–cause), while x means "no FI is detected". For clarity, we also use colors to indicate cl (yellow), cr (green) and cc (blue). For brevity, some detected FIs have not been presented; they are indicated with * in Table 1. Considering that the same FI can be named S_1–S_2 and S_2–S_1, Table 1 is a symmetric matrix; that is why only a triangular half of Table 1 is specified.

4.5. New FIs

Our Cause–Restrict approach has been validated for the detection of several FIs known and documented in the literature, like all FIs not using services CC, FM, BLF, PIN and ML. We have also detected new FIs, i.e., FIs which we were unaware of (we did not find them in the literature). These new FIs are those using services CC, FM, BLF, PIN and ML. Some of these new FIs have been presented in detail in Section 4 (FM–TCS) and Appendix B (CC–OCS, ML–BLF, PIN–TCS). Other new FIs have been mentioned as resembling to the FIs presented in detail, like CC-TCS which resembles CC–OCS, and FM–OCS which resembles FM–TCS. Those resembling FIs CC–TCS and FM–OCS can be easily deduced from the original FIs CC–OCS and FM–TCS, by using the analogy between OCS and TCS. Other new FIs have not been presented for brevity. They are indicated by * in Table 1. Let us present their principle intuitively and briefly.

Interactions ML-S, where S equals to AR, CW, or UM

ML assumes that his subscriber can be at the same time not idle and not busy, when some (but not all) of his lines are busy. The other services (AR, CW, UM) assume that B is idle if and only if he is not idle. Let S be one of these three services and assume that B is a subscriber of ML and S. If B has one of his lines busy and receives a call from A, then, should service S be triggered (by considering B busy, according to S) or not (by considering B as not busy, according to ML)?

Interactions FM-S, where S equals to CFU, UM, or Email

FM assumes that if his subscriber receives a call and does not answer and is not busy, then another number is tried. The other services (CFU, UM, Email) assume that another action A (different from trying another number) is executed when B receives a call. Let S be one of these three services and assume that B is a subscriber of FM and S. If B receives a call from A and does not answer, should action A be executed (according to S) or should another number be tried (according to FM)?

Interactions PIN-S, where S equals to AR or BLF

When a subscriber A of PIN calls a subscriber B of S from the phone of a user C, it is not clear whether S should behave with respect to A or to C. For example, should AR recall A or C? And should BLF display information on A or on C?

5. Related work

To our best knowledge, Chentouf et al. (2004) and Kolberg and Magill (2007) are two approaches that are closely related to our approch. Let us, therefore, make a comparative analysis of our contributions with these references.

Chentouf et al. (2004) and Kolberg and Magill (2007) had the research objective to come out with a service modeling language that abstracts service details. Both research works proposed abstract languages and associated FI detection methods that are based on a syntactical comparison of service models and FI detection rules that incarnate pre-defined FI patterns. Unknown FI cannot be detected by those two approaches unless the corresponding FI detection rules are set. Our FI detection method is semantics-based. It abstracts all the actions into two: cause and restrict, and it relies on three fixed FI patterns that will not need to be updated as new services are added: cause–loop, cause–cause, and cause–restrict. This fundamental difference between our work and the two related ones clearly shows that ours is better than the two other approaches.

A second difference between our proposed approach and the two related ones consists in the expressiveness of the service modeling languages and the effectiveness of the associated FI detection methods. In fact, the language proposed by Chentouf et al. (2004), called Feature Interaction Management Language (FIML), cannot model CC, PIN, BLF, UM, and ML. The language of Kolberg and Magill (2007) cannot capture CC, PIN, BLF, UM, ML, FM, Email, and F-Email. Consequently, the other two approaches cannot detect FI among those services.

A third aspect of comparison between our work and the two related ones is the abstraction level of the modeling language. As already explained, our approach is semantic while the two related works are syntactical. If two service operators networks need to interoperate, they have to adopt the CR-language. They only need to exchange the behavior descriptions of their services; the interface descriptions do not have to be communicated. Compared with FIML, the behavior part of the CR-language is situated at the same level of abstraction as both languages contain the same concepts: user, address, call; events like busy, no answer, etc.; and actions such as hold, email, etc. However, in FIML, every service behavior statement has to be written under a specific processing point. A processing point indicates the step of the call processing, for example, dialing, response received, call established, etc. This level of details is not revealed in the CR-language, and hence the CR-language is more abstract than FIML.

The work of Kolberg and Magill (2007) contains two parts: the first part defines a modeling language and the second part adaptes the language to SIP, which is a signaling protocol (Rosenberg et al., 2002). The first part can be exploited as an offline FI solution. The second part is meant to be executed at runtime.

Compared with the language proposed by Kolberg and Magill (2007), the CR-language appears to be less abstract at a first glance. Their proposed language describes the connections the caller's device might establish with other user terminals during the call. There are two types of connections: the original connection that is supposed to be set up, and the effective connection that is set up after the service has been executed. Service models also contain the so called *treatments*. A treatment is any processing that is run in the network, i.e., in any server that is involved in the ongoing call. Such a processing is triggered by an event that may be call-related like busy tone, or not call-related like network congestion. By introducing this concept, the authors aimed at abstracting all kinds of events in the concept of treatment. However, abstracting all events and kinds of processing that a server may run in only one concept has a serious side effect: as formulated, the treatment is vague. This engenders a FI detection drawback. In fact, if a service execution results in connecting the caller address to a treatment instead of the targeted callee address, the detection procedure concludes that there is a FI. The article does not explain how the detection procedure can distinguish a final treatment, after which the caller will never be connected to the callee, from intermediary treatments, after which the connection between the caller and the callee can be established. We conclude that the language proposed by Kolberg and Magill is too abstract to the extent that the fundamental concept of treatment is vague and might engender ambiguity.

Although our work presents an offline FI solution, we think that analyzing the online part of Kolberg and Magill work is interesting. Kolberg and Magill applied their approach to SIP. They then explained how the solution can be exploited at runtime. In this part of their work, treatment becomes clearer as they consider any SIP response and the processing that it may trigger in a server as a treatment. However, the authors do not explain how to distinguish between the two types of treatments (intermediary and final). We think that the distinction should be based on the SIP response types: intermediary treatments are triggered by intermediary SIP responses, i.e., 1xx, 2xx, and 3xx messages, and final treatments are fired by final SIP responses, i.e., 4xx, 5xx, and 6xx responses (Rosenberg et al., 2002). However, some exceptions to this rule have to be carefully examined. Indeed, 200 OK SIP response message, for example, sometimes causes the session termination depending on the preceding SIP request(s). The 407 Proxy Authentication Required does not terminate the call; it simply requires the caller's device to send its authentication credentials. We conclude that to be exploitable on line (i) the approach of Kolberg and Magill needs to be tailored to the underlying signaling protocol, (ii) the proposed SIP implementation of the approach needs to be refined based on the SIP response types.

Another advantage the CR-language has over the two related works emanates from the fact that it is object-oriented. We think that the derivation of CR-models from object oriented analysis documents (UML diagrams, for example), can be automated.

6. Conclusion

We have first developed a Cause–Restrict language to model subscribers of telecommunication services at a high abstraction level. A Cause–Restrict model of a subscriber provides information such as: what is the cause of what, and what restricts what, and specifies coarsely the frequency of each operation "cause" or "restrict" by "always" or "sometimes". Then, we have developed a method that detects feature interactions between telecommunication services modeled in the Cause–Restrict language. The latter has permitted to reduce the state space explosion encountered in several feature interaction detection methods. We have demonstrated the applicability of our approach for the description of several services and the detection of several feature interactions between them. Known FIs and, more interestingly, new FIs have been detected.

As a future work, we plan to study the feature interaction *resolution* phase, which consists in finding solutions to the detected feature interactions. Both of the online and off-line resolutions will be investigated.

Appendix A.

A.1. Subscriber of Originating Call Screening (OCS)

A subscriber of OCS registers users in a list L_{OCS} so that every outgoing call from A toward a user registered in L_{OCS} is automatically blocked (note the symmetry with TCS).

<u>Interface model</u>: subscriber of OCS	
userOCS extends user	
// Attributes	
users *ListOcs*	// Specific attribute corresponding to L_{OCS}
	// of the current subscriber of OCS
// Methods	
call *call*(**user**)	// Overrides the method *call*() of the class **user**

The method *call*() of **user** is overridden because calls to users registered in L_{OCS} must not be established.

CR-behavior model: subscriber A of OCS

The following CR-relations are added to the basic behavior:

$(B \in A.ListOcs) \wedge A.call(B)$ restrict! $A.anyCom(B)$

$(B \in A.ListOcs) \wedge A.call(B)$ restrict! $B.anyCom(A)$

$(B \in A.ListOcs) \wedge (A = Call.Initiator)$ restrict! $B \in Call.Participants$

A.2. Subscriber of email

If a subscriber A of Email calls a user B, an email is automatically sent from A to B. Such an email may contain advertisement, for example.

```
Interface model: subscriber of Email
userEmail extends user
// Attributes: no new attribute is defined
// Methods
call call(user)                // Overrides the method
                               call() of the class user.
void email(user, text)         // New method: it sends a
                               text message to a user.
                               // The user and message
                               // are given as parameters.
```

The method *call*() of **user** is overridden because Email adds an email sending with every call. A new method *email*() is added.

CR-behavior model: subscriber A of Email

The following CR-relation is added to the basic behavior:

$A.call(B)$ cause! $A.email(B,\ \text{text})$

A.3. Subscriber of F-email

A subscriber of F-email can forbid the reception of emails which come with calls.

```
Interface model: subscriber of F-email
userFemail extends user
// Attributes
boolean forbidEmail            // Generic attribute
                               // indicating whether reception
                               // of emails is forbidden,
                               // i.e., if service is enabled.
                               // We consider here
                               //only emails that come with
                               // calls, not all the emails.
// Methods
void receiveEmail(user, text)  // New method: it receives an
                               // email with a call from a user.
```

A new method *receiveEmail*() is added.

CR-behavior model: subscriber B of F-email

The following CR-relation is added to the basic behavior:

$A.call(B) \wedge (B.forbidEmail = \text{true})$ restrict! $B.receiveEmail(A, anyMessage)$

A.4. Subscriber of Conference Call (CC)

A subscriber A of CC can ask his provider's server to program a conference call at a given future time T. The server sends a phone number N and a password P to A who forwards this information to users he wants to invite to the conference call. Any user (including A) that knows (N, P) can join the conference at time T by calling N and then entering P. A is considered the initiator of that conference call.

```
Interface model: subscriber of CC
userCC extends user
// Methods
call program()                 // New method: program a
                               // conference call
void join(call)                // New method: join a
                               // programmed conference call.
```

Two new methods are defined in **userCC**: *Call* = *A.program*() means that A programs a conference call, *A.join*(*Call*) means that A joins a conference call.

CR-behavior model: subscriber A of CC

The following CR-relations are added to the basic behavior:

$(Call = A.program()) \wedge A.join(Call)$ cause! $A = Call.Initiator$

$Call.accept(X)$ cause! $X = Call.Participants$

A.5. Subscriber of PIN-Calling (PIN)

This is an office service where some privileged employees have the right to use some services. For that, they have to dial a PIN (Personal Identification Number). If a PIN owner A uses the phone of a colleague

B, A has access to all his services even if B, the phone owner, has not the right to use those services.

```
Interface model: subscriber of PIN
userPIN extends user
// Attributes
user phoneOwner                // Owner of the phone used
                               // by the subscriber of PIN.
```

CR-behavior model: subscriber A of PIN

The following CR-relations are added to the basic behavior:

$(A.phoneOwner = C) \wedge (Call = A.call(B))$cause!$C.anyCom(B)$

$(A.phoneOwner = C) \wedge (Call = A.call(B))$ cause!$B.anyCom(C)$

A.6. Subscriber of Busy Lamp Field (BLF)

The service Presence permits to users and servers to know the status of other users. BLF is a specific service that needs Presence in the following way. A subscriber A of BLF can specify a list of users for whom he wants to watch the status: idle or busy. If A watches the status of B, every time B makes or receives a call, A's phone displays the information that B is busy. If B is not on the phone, A's phone displays the information that B is idle.

```
Interface model: subscriber of BLF
userBLF extends user // Attributes
user watchee                 // Watchee is the user watched by
                             // the BLF service subscriber.
boolean watcheeBusy          // True means that watchee is
                             // busy, false means he is idle.
```

CR-behavior model: subscriber A of BLF

The following CR-relations are added to the basic behavior:

$(A.watchee = B) \wedge (B.busy = \text{true})$ cause!$(A.watcheeBusy = \text{true})$
$(A.watchee = B) \wedge (B.busy = \text{false})$ cause!$(A.watcheeBusy = \text{false})$
$(A.watchee = B) \wedge (A.watcheeBusy = \text{true})$ cause!$(B.busy = \text{true})$
$(A.watchee = B) \wedge (A.watcheeBusy = \text{false})$ cause!$(B.busy = \text{false})$

A.7. Subscriber of Multiple Lines (ML)

A subscriber A of ML has several lines (say N) in the same phone sharing the same number. A can accept a new call while he is already on the phone. And so on, A can accept other calls until all the N lines are busy.

```
Interface model: subscriber of ML
userML extends user
// Attributes
int busyLines                          // Number of busy lines
```

CR-behavior model: subscriber B of ML

The CR-relation 5b (($B.idle$ = false) cause! ($B.busy$ = true)) of the basic behavior is removed from the basic behavior, and the following CR-relations are added:

$(B.busyLines < N)$ restrict! $B.busy = \text{true}$,
$(B.busyLines = N)$ cause! $B.busy = \text{true}$.

Appendix B.

B.1. OCS–CFU

Context: Consider A and B who are subscribers to OCS and CFU respectively, and a third user C. Assume that A has put C in his list L_{OCS}, with the idea that a call initiated by A has no effect on C (hence A is not authorized to join C). Assume that B has programmed an automatic unconditional forward of his incoming calls towards C.

Scenario of FI: OCS prevents that A joins directly C. But if A calls B, A will be forwarded to C (and hence A joins C) although C is in L_{OCS}.

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of CFU that has programmed a forward towards C, we have by using the 2nd CR-relation of Section 2.4.2:

$$\text{M1} : (B.forward = C) \wedge (Call = A.call(B)) \text{ cause!} C \in Call.Participants$$

– For the subscriber A of OCS that has put C in L_{OCS}, we obtain by adapting the 3rd CR-relation of Appendix A.1:

$$\text{M2} : (C \in A.ListOcs) \wedge (A = Call.Initiator) \text{ restrict! } C \in Call.Participants$$

The pair of CR-relations (M1, M2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where the participation of C in a call is at the same time implied by CFU and forbidden by OCS (see the above *Context-Scenario*).

We obtain resembling FIs if we replace CFU by CFBL (Call Forward on Busy Line), CFNR (Call Forward on No Reply), or CFT (Call Forward on Time). Another resembling FI is TCS–CFU (studied in Section 4.2.2), which is obtained if we replace "A is subscriber of OCS" by "C is subscriber of TCS".

B.2. TCS–CFU

Context: Consider B and C who are subscribers to CFU and TCS respectively, and a third user A. Assume that C has put A in his list L_{TCS}, with the idea that a call involving C but not initiated by C has no effect on A (hence C is not authorized to be joined by A). Assume that B has programmed an automatic unconditional forward of his incoming calls toward C.

Scenario of FI: TCS prevents that C is directly joined by A. But if A calls B, A will be forwarded to C (and hence C is joined by A) although A is in L_{TCS}.

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of CFU that has programmed a forward toward C, we obtain by adapting the 2nd CR-relation of Section 2.4.1:

$$\text{M1} : (B.forward = C) \wedge (Call = A.call(B)) \text{ cause!} C \in Call.Participants$$

– For the subscriber C of TCS that has put A in L_{TCS}, we obtain by adapting the 4th CR-relation of Section 2.4.2:

$$\text{M2} : (A \in C.ListTcs) \wedge \ (A = Call.Initiator) \text{ restrict! } C \in Call.Participants$$

The pair of CR-relations (M, M2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where the participation of C in a call is at the same time implied by CFU and forbidden by TCS (see the above *Context-Scenario*).

We obtain a resembling FI if we replace CFU by Call Forward on Busy Line (CFBL), Call Forward on No Reply (CFNR), or Call Forward on Time (CFT).

B.3. TCS–UM

Context: Consider two users A and B, where B is subscriber to TCS and UM. Assume that B has put A in his list L_{TCS}, with the idea that a call involving B but not initiated by B has no effect on A (hence B is not authorized to be joined by A).

Scenario of FI: TCS prevents that B is directly joined by A in a basic call. But if A calls B who is busy or does not answer, B receives by email a voice message from A. Hence, the call initiated by A has an effect on B, contrary to the aforementioned idea of B when he has put A in L_{TCS}.

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of TCS has put A in L_{TCS}, we have by using the 2nd CR-relation of Section 2.4.2:

$$\mathrm{M1} : (A \in B.ListTcs) \wedge A.call(B) \text{ restrict! } B.anyCom(A)$$

– For the subscriber B of UM who receives a call from A, we have by using the CR-relation of Section 2.4.5:

$$\mathrm{M2} : (B.um = \text{true}) \wedge A.call(B) \wedge (B.busy = \text{true} \vee B.noAnswer = \text{true}) \text{ cause! } B.unifiedMessaging(A, \ voiceMsg)$$

If in M1 we replace *anyCom*(*A*) by *unifiedMessaging* (*A*, “I called you but I did not answer.”), the pair of CR-relations (M1, M2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where a communication of B with A is at the same time implied by UM and forbidden by TCS (see the above *Context-Scenario*).

We obtain a resembling FI TCS–Email if we replace “B is subscriber of UM” by “A is subscriber of Email”. Another resembling FI is OCS–UM, which is obtained if we replace “B is subscriber of TCS” by “A is subscriber of OCS”. Yet another resembling FI is OCS–Email which is studied in the following subsection B4. Since UM contains the service Voicemail (VM), we can obtain more FIs if we replace “B is subscriber of UM” by “B is subscriber of VM”. Hence the interactions OCS-VM and TCS-VM.

B.4. OCS–Email

Context: Consider two users A and B, where A is subscriber to OCS and Email. Assume that A has put B in his list L_{OCS} with the idea that a call initiated by A has no effect on B (hence A is not authorized to join B).

Scenario of FI: OCS prevents that A joins directly B in a basic call. But if A calls B, A sends an email message to B (hence A joins B) although B is in L_{OCS}.

Let us now show how this FI is detected by our FI detection method:

– For the subscriber A of OCS that has put B in L_{OCS}, we have by using the 1st CR-relation of Appendix A.1:

$$\mathrm{M1} : (B \in A.ListOcs) \wedge A.call(B) \text{ restrict! } A.anyCom(B)$$

– For the subscriber A of Email that calls B and sends him an automatic email to B, we have by using the CR-relation of Appendix A.2:

$$\mathrm{M2} : (A.call(B)) \text{ cause! } A.email(B, emailMsg)$$

If in M1 we replace *anyCom*(*B*) by *email*(*B*, “hello, I am calling you.”), the pair of CR-relations (M1,M2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where a communication of A with B is at the same time implied by Email and forbidden by OCS (see the above *Context-Scenario*).

We obtain a resembling FI if we replace Email by EBL (Email on Busy Line) where the caller sends an email only if the callee is busy, for example to inform him that he has called him.

B.5. Email–F-email

Context: Consider A and B who are subscribers to F-email and Email respectively. Assume that A has forbidden the reception of emails.

Scenario of FI: the FI is due to contradictory objectives of Email and F-email: If B calls A, should B send an email message to A (according to Email) or should not he (according to F-email) ?

Let us now show how this FI is detected by our FI detection method:

– For the subscriber B of Email that calls A and sends him an automatic email to A, we obtain by adapting the CR-relation of Appendix A.2:

$$\mathrm{M1} : (B.call(A)) \text{ cause! } B.email(A, emailMsg)$$

– For the subscriber A of F-email that forbids reception of emails, we obtain by adapting the CR-relation of Appendix A.3:

$$\mathrm{M2} : B.call(A) \wedge (A.forbidEmail = \text{true}) \text{ restrict! } A.receiveEmail(B, anyMessage)$$

– The following CR-relation is added by the designer (Step 1b) to state that the sending by B of an email toward A is followed by the reception by A of an email from B:

$$\mathrm{M3} : B.email(A, emailMsg) \text{ cause! } A.receiveEmail(B, emailMsg)$$

By applying rule r5 to M1 and M3, we obtain:

$$\mathrm{N1} : (B.call(A)) \text{ cause! } A.receiveEmail(B, emailMsg)$$

The pair of CR-relations (N1, M2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where the transmission of an email from B to A is at the same time implied by Email and forbidden by F-email (see the above *Context-Scenario*).

We obtain a resembling FI EEC–F-email if we replace Email by EEC (Email on end of call) where a call participant sends an email when he terminates a call. Another resembling FI UM–F-email, which is obtained if we replace “A subscriber of Email” by “B subscriber of UM”.

B.6. OCS–CC

Context: Consider a subscriber A of OCS and CC. Assume that A has put B in his list L_{OCS} with the idea that a call initiated by A has no effect on B (hence A is not authorized to join B).

Scenario of FI: OCS prevents that A joins directly B in a basic call. But if A programs a conference call which is joined by A and B, A can join B although B is in L_{OCS}.

Let us now show how this FI is detected by our FI detection method:

- For the subscriber A of CC that programs a conference which is joined by A and B, we have by using the CR-relations of Appendix A.4:

$$\text{M1} : (\mathit{Call} = \mathit{A.program}()) \wedge \mathit{A.join}(\mathit{Call}) \text{ cause!} A = \mathit{Call.Initiator}$$
$$\text{M2} : \mathit{Call.accept}(X) \text{ cause!} X = \mathit{Call.Participants}$$

- For the subscriber A of OCS that has put B in L_{OCS}, we have by using the 3rd CR-relation of Appendix A.1:

$$\text{M3} : (B \in \mathit{A.ListOcs}) \wedge (A = \mathit{Call.Initiator}) \text{ restrict! } B \in \mathit{Call.Participants}$$

By applying rule R7 To M1 and M3 with U = "(*Call* = *A.program*()) ∧ *A.join*(*Call*)", V = "*A* = *Call.Initiator*", W = " *B*∈ *Call.Participants*", Z = "*B*∈*A.ListOcs*", we obtain:

$$\text{N1} : (B \in \mathit{A.ListOcs}) \wedge (\mathit{Call} = \mathit{A.program}()) \wedge \mathit{A.join}(\mathit{Call}) \text{ restrict! } B \in \mathit{Call.Participants}$$

If we replace X by B in M2, the pair of CR-relations (M2, N1) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where the participation of B in a call is at the same time implied by CC and forbidden by OCS (see the above *Context-Scenario*).

We obtain a resembling FI if we replace "A subscriber of OCS" by "B subscriber of TCS".

B.7. BLF–ML

Actually, this FI involves also the Presence service. The latter is implicit in BLF–FM because BLF necessitates Presence (see Appendix A.6).

Context: Consider A and B who are subscribers of BLF and ML, respectively, and a third user C. Presence is designed so that the status busy is set to true when the watched user is on the phone. Hence, a subscriber of BLF will see the "watchee" as busy when he is on the phone.

Scenario of FI: A calls B who is on the phone with C. B is seen as busy according to BLF, while he is considered as idle according to ML. In other words, BLF and ML do not use the same semantic for *B.busy*.

Let us now show how this FI is detected by our FI detection method:

- For the subscriber A of BLF, we have by using the 3rd CR-relation of Appendix A.6 and the CR-relation 5b of Section 2.3:

$$\text{M1} : (\mathit{A.watchee} = B) \wedge (\mathit{A.watcheeBusy} = \text{true}) \text{ cause!}(\mathit{B.busy} = \text{true})$$
$$\text{M2} : (\mathit{B.idle} = \text{false}) \text{ cause! } (\mathit{B.busy} = \text{true})$$

- For the subscriber B of ML, we have by using the CR-relation of Appendix A.7:

$$\text{M3} : (\mathit{B.busyLines} < N) \text{ restrict! } (\mathit{B.busy} = \text{true})$$

The pairs of CR-relations (M1, M2) and (M1, M3) constitute cause–restrict patterns which are symptoms of conflict. We have checked that we can reach a situation where at the same time B is seen is busy according to BLF while he is seen as non busy according to ML (see the above *Context-Scenario*).

B.8. PIN–TCS

Context: Consider A and B who are subscribers of PIN and TCS, respectively, and a third user C. Assume that B has put C in his list L_{TCS}, with the idea that C is not authorized to be joined by B. PIN is designed so that if A uses C's phone to enter his PIN and calls B, the latter sees A as C.

Scenario of FI: A uses C's phone to enter his pin and calls B. The call of A is blocked although A is not in L_{TCS}, because the phone used by A (C's phone) is in L_{TCS}.

Let us now show how this FI is detected by our FI detection method:

- For the subscriber A of PIN that uses C's phone to call B, we have by using the 1st CR-relation of Appendix A.5:

$$\text{M1} : (\mathit{A.phoneOwner} = C) \wedge (\mathit{Call} = \mathit{A.call}(B)) \text{ cause! } \mathit{C.anyCom}(B)$$

- For the subscriber B of TCS that has put C in L_{TCS}, we obtain by adapting the 1st CR-relation of Section 2.4.2:

$$\text{M2} : (C \in \mathit{B.ListTcs}) \wedge (\mathit{C.call}(B)) \text{ restrict! } (\mathit{C.anyCom}(B))$$

The pair of CR-relations (M1, M2) constitutes a cause–restrict pattern which is a symptom of conflict. We have checked that we can reach a situation where TCS blocks a call which, according to PIN, should not be blocked (see the above *Context-Scenario*).

B.9. AR–UM

Context: Consider two users A and B, where B is subscriber to AR and UM.

Scenario of FI: If A calls a busy user B, B receives an email from A containing a voice message (according to UM) while A is recalled back later by B (according to AR). Which of the two actions should be executed? Should we execute both actions?

Let us now show how this FI is detected by our FI detection method:

- For the subscriber B of AR that has programmed an automatic recall and receives a call from A while he is busy, we have by using the CR-relation of Section 2.4.3:

$$\text{M1} : (\mathit{B.ar} = \text{true}) \wedge (\mathit{Call} = \mathit{A.call}(B)) \wedge (\mathit{B.busy} = \text{true}) \text{ cause! } \mathit{B.call}(A) \wedge (B \in \mathit{Call.Participants} \setminus \{\mathit{Call.Initiator}\}) \wedge (A = \mathit{Call.Initiator})$$

- For the subscriber B of UM that receives a call from A and then sends him a voicemail message, we have by using the CR-relation of Section 2.4.5:

$$\text{M2} : (\mathit{B.um} = \text{true}) \wedge \mathit{A.call}(B) \wedge (\mathit{B.busy} = \text{true} \vee \mathit{B.noAnswer} = \text{true}) \text{ cause! } \mathit{B.unifiedMessaging}(A, \text{voiceMsg})$$

The pair of CR-relations (M1, M2) constitutes a cause–cause pattern.

We obtain a resembling FI if we replace "B subscriber of UM" by "B is subscriber of VM" or "A subscriber of Email". We have checked that we can reach a situation where B calls A according to AR while B receives a voice message by email according to UM (see the above *Context-Scenario*).

References

Amer, M. et al., 2000. Feature interactions resolution using fuzzy policies. In: [5], pp. 94–112.

Amyot, D. et al., 2000. Feature description and Feature Interaction analysis with Use Case Maps and LOTOS. In: [5], pp. 252–261.

Amyot, D., Logrippo, L. (Eds.), 2003. Feature Interactions in Telecommunications and Software Systems VII. IOS Press, Amsterdam.

Blom, J. et al., 1994. Using temporal logic for modular specification of telephone services. Feature Interactions in Telecommunications Systems. In: [1], pp. 197–216.

Blom, J. et al., 1995. Automatic detection of feature interactions in temporal logic. In: [5], pp. 1–19.

Bouma, L.G., Velthuijsen, H. (Eds.), 1994. Feature Interactions in Telecommunications Systems. IOS Press, Amsterdam.

Calder, M., Magill, E. (Eds.), 2000. Feature Interactions in Telecommunications and Software Systems VI. IOS Press, Amsterdam.

Calder, M. et al., 2007. Hybrid solutions to the feature interaction problem. In: [8], pp. 295–312.

Charnois, T., 1997. A natural language processing approach for avoidance of feature interactions. In: [3], pp. 347–363.

Cheng, K.E., Ohta, T. (Eds.), 1995. Feature Interactions in Telecommunications Systems III. IOS Press, Amsterdam.

Chentouf, Z. et al., 2004. Service interaction management in SIP user device using Feature Interaction Management Language. In: Proceedings of the 2004 Conference Nouvelles Technologies de la Repartition (NOTERE), Morocco.

Cherkaoui, S., Khoumsi, A., 2002. Mobile and static agents for service interactions resolution in telecommunication environments. In: 9th IEEE International Conference on Telecommunications (ICT'2002), Beijing.

Dini, P. et al. (Eds.), 1997. Feature Interactions in Telecommunication Networks IV. IOS Press, Amsterdam.

Du Bousquet, L., Richier, J.-L. (Eds.), 2007. Feature Interactions in Software and Communication Systems IX. IOS Press.

Gammelgaard, A., Kristensen, J.E., 1994. Interaction detection, a logical approach. In: [1], pp. 178–196.

Gibson, P., Mery, D., 1997. Telephone feature verification: translating SDL to TLA+. In: Eighth SDL Forum (SDL'1997), Evry, France.

Griffeth, N.D., Velthuijsen, H., 1994. The negotiating agents approach to runtime interaction resolution. Feature Interactions in Telecommunications Systems. In: [1], pp. 217–235.

Khoumsi, A., 1997. Detection and resolution of interactions between services of telephone networks. In: [3], pp. 78–92.

Kimbler, K., Bouma, L.G. (Eds.), 1998. Feature Interactions in Telecommunications and Software Systems V. IOS Press, Amsterdam.

Kolberg, M., et al., 2002. Feature Interactions in Services for Internet Personal Appliances. In: Proceedings of IEEE International Conference on Communications (ICC 2002), New York, pp. 2613–2618.

Kolberg, M., Magill, E., 2007. Managing feature interactions between distributed SIP call control services. Computer Networks 51, 536–5575.

Nakamura, M. et al., 1997. Petri-net based detection method for non-deterministic feature interactions and its experimental evaluation. In: [3], pp. 138–152.

Nakamura, M., Reiff-Marganiec, S. (Eds.), 2009. Feature Interactions in Software and Communication Systems X. IOS Press.

Reiff-Marganiec, S., Ryan, M. (Eds.), 2005. Feature Interactions in Telecommunications and Software Systems VIII. IOS Press, Amsterdam.

Rosenberg, J. et al., 2002. SIP: Session Initiation Protocol. IETF RFC 3261.

Tsang, S., Magill, E.H., 1997. Behavior based run-time feature interaction detection and resolution approaches for intelligent networks. In: [3], pp. 254–270.

Weiss, M. et al., 2007. Towards a classification of web service feature interactions. Computer Networks 51, 359–381.