The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage

Nicolas Gorse

Thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

under the auspices of the
Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
September, 2000

Abstract

This thesis addresses the feature interaction problem in the research area of telephony systems design. We start with a review of this problem and present our definition of a feature and of a feature interaction. We enumerate and discuss some known approaches with respect to their strengths and limitations.

We present our initial approach and propose some refinements in order to automate the detection of incoherences and derivation of validation scenarios at the design stage using predicate logic and Prolog. The identification of incoherences corresponding to potential feature interactions allows the designer to refine the requirements and to produce a better specification.

A UCM model is derived from requirements. This model allows the designers to have a general point of view of the system and to detect design defaults. The construction of the UCM model is followed by the generation of a LOTOS specification.

The derivation of validation test suites and their application against the specification permits to check whether or not the incoherences identified in the requirements lead to feature interactions.

The results obtained over two case studies show that the application of this approach improves the design and validation of a specification.

Acknowledgement

I would like to thank all the people who helped me in completing this thesis.

First, I would like to thank my supervisor, Dr. Luigi Logrippo, for his encouragement, guidance and assistance through my graduate studies.

Second, I would like to thank the members of the University of Ottawa LOTOS group for their advises and helpful comments. I would like to thank particularly Jacques Sincennes who suggested the idea of using predicate logic and Prolog to identify incoherences and who supported me through long discussions.

Third, I would like to thank for their financial support: the University of Ottawa, Mitel Corporation, and Communications and Information Technology Ontario (CITO).

Last but not least, I dedicate this thesis to my wife, Sandra Jacqueson, who always supported me and listened patiently to all my endless monologues.

Contents

Chapter 1

Introduction

1.1 Background

Specification and testing of complex distributed systems have always been a challenge. Today's telephone systems are often decomposed into a base system *(the kernel)* and features [1, 2]. The base system implements the basic behavior, such as simple calls management, while features are optional units or increments of functionality that are available in the system and that can be used to implement more complicated behaviors such as *Call Waiting* or *Call Forward on Busy Line*. A user can then choose to subscribe to the use of some of these features depending on his/her needs.

The advantage of such *Feature-Oriented* systems is that they separate the principal behavior from other optional extended ones. This distinction facilitates the addition of new capabilities without having to redesign or re-implement the whole system. However, the combination of features can lead to conflicting or contradictory behaviors. This problem, the so-called *feature interaction* problem [3, 4, 1], is characterized by the fact that some features can be combined together in such a way that they result in non-desired behaviors.

Some interactions are desirable and planned. Others are to be avoided, and most of these can be avoided at the specification and validation stages. Specification and validation are two major phases taking place in the design of complex, and usually distributed, architectures such as telephony systems. The design usually starts in the early stages with the informal requirements followed by formal specifications [5]. Once the system is formally specified, a validation can be performed in order to verify, as far as possible, that the specification is free of errors. Such a validation also insures that the specification of the system fulfills the requirements.

1.2 Motivation

Various Formal Description Techniques [6, 7, 8] address the verification and validation of systems. These techniques can be directly applied to the feature interaction problem but their efficiency can be sometimes limited. Looking for possible feature interactions at the validation level often implies checking each and every possible combination of features, in all possible execution sequences. Even if only considering pair-wise combinations, the number of combinations to analyze can be considerably large. Hence, trying to find feature interactions in a detailed specification of a system is not always a viable option.

A help to find interactions is to identify, at the requirements level, combinations presenting specific *incoherences* that can lead to potential interactions in the specification.

In the reminder of this thesis, the concept of incoherence will play a crucial role. This concept will be defined formally in chapter 4, section 4.2. For the time being, we provide the following informal definition: an incoherence refers to some incoherent behavior of features with respect to what would be coherent from the point of view of the intended system's behavior.

An example of such an incoherence is the fact that two features, let us say `Incoming Call Screening` and `Call Forward on Busy`, can be triggered by the same event and present different results (see chapter 4, section 4.2.2). Another example is the fact that a feature such as `Call Forward on Busy`, results in triggering another feature such as `Outgoing Call Screening` while their results present a contradiction (see chapter 4, section 4.2.3).

Features can be formally modeled using an appropriate formal representation and information provided by the requirements. The use of a formal representation allows the application of formal filtering rules to identify incoherences between features.

Then, a scenario based validation can be performed using automatic derivation of test suites: for each incoherence identified, a validation test suite can be built and applied to the specification to check whether or not the incoherence leads to an interaction. Using such a method allows to focus on the potential interactions and hence to reduce the testing phase by restricting tests referring to combinations that do present incoherences.

1.3 The Proposed Approach

Based on the ideas presented above, we propose an approach targeting the identification of incoherences corresponding to potential interactions and the validation of a specification with respect to such incoherences. Our work does not address run-time resolution of feature interactions [3, 4, 9, 10]. The identification of incoherences and detection of feature interactions is done statically during the specification phase of the system. The proposed approach is mainly composed of a filtering part and a validation part.

1.3.1 Filtering

Starting from requirements, the filtering part consists in identifying incoherences between feature specifications. Based on the information provided by requirements, features are modeled using a formal representation. The features modeled are analyzed by a tool that implements formal rules for the identification of incoherences between pairs of features. From the requirements and knowledge obtained from this filtering, a formal specification is derived. The knowledge provided by filtering allows to derive a better specification by making the designers aware of the incoherences present in requirements.

1.3.2 Validation

Based on the knowledge obtained from the filtering, the validation part consists in checking whether or not the incoherences previously identified lead to interactions in the specification.

The correspondence between the representation of features for the filtering and their representation in the specification is done via *mapping rules*. These mapping rules are used to translate the incoherences identified by the filtering tool into test suites. Such test suites are applied against the specification to perform scenario based validation, targeting the detection of feature interactions corresponding to the incoherences identified.

1.4 Organization of the Thesis

The thesis consists of an introduction, five main chapters and a conclusion. Each chapter is presented below with a brief summary of its contents.

Chapter 1 – Introduction

Chapter 1 contains the introduction of the thesis.

Chapter 2 – Framework

Chapter 2 describes the conceptual framework on which this thesis is based. It introduces the reader to the so-called feature interaction problem and some of the existing approaches and their limitations. The structure of our research project is discussed, along with our filtering and validation approach.

Chapter 3 – Use Case Maps, LOTOS & Mitel's System

Chapter 3 presents the specification part of our project. It introduces the reader to Use Case Maps, their purpose and notation and the telephony system (provided by Mitel Corp. as a set of UCM) on which this work is based. In addition, the LOTOS language and the translation principles of a UCM model into LOTOS are presented.

Chapter 4 – Identification of Incoherences – Theory

Chapter 4 presents the proposed filtering process. This process is based on a formal representation of features and on formal *incoherence rules* targeting the identification of incoherences corresponding to potential interactions between features. The basic principles of our feature representation, formalism and incoherence rules are presented.

Chapter 5 – Identification of Incoherences & FI Detection – Tool

Chapter 5 presents the scenario based validation of our LOTOS specification with respect to features and feature interactions testing. A tool for automatic filtering of incoherences and automatic derivation of validation test suites is presented.

Chapter 6 – Application & Evaluation

Chapter 6 presents the application of our approach on two different specifications. In addition, we present a performance evaluation of our tool and a comparison with a similar technique.

Chapter 7 – Conclusion & Future Work

Chapter 7 concludes the thesis. It reviews the achievements and contributions and presents the possible improvements of the approach along with further research.

Chapter 2

Framework

This chapter introduces the reader to the feature interaction problem and to some of the current approaches used to deal with it. It also describes the project to which our work is related and our initial approach, and refinements, for the detection and validation of feature interactions. The approach is developed and explained in detail along the thesis.

2.1 The Feature Interaction Problem

The feature interaction problem is often related to telephony system features. However, a feature interaction can occur in any complex system and the definition of such interactions need to be clearly stated. This section presents our definition of a feature as well as what we consider to be a feature interaction with respect to telephony systems. Having introduced the problem, we give reasons of its prominence in such systems and we review some existing approaches. Then, we propose an approach for the detection and validation of feature interactions in the early stages of development of a system.

2.1.1 Definition of a Feature

The concept of feature in telephony is best described pragmatically with respect to user requirements. A feature is a functionality offered by a system and has the purpose of fulfilling certain user intentions in the context of a call. A feature is seen as an optional unit or increment of functionality that is added to a system [1]. Such a 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 $\mathcal{F}_x$ where $1 < x < n$ is denoted: $\mathcal{B} \oplus \mathcal{F}_1 \oplus \mathcal{F}_2 \oplus ... \oplus \mathcal{F}_n$ [11, 12].

This way of modeling separates the behavior of the base of the system from the behavior of the features. As well, it separates single feature behaviors from composed feature behaviors. We believe that one advantage of such modeling is that it makes the specification usually easier to extend, modify, and test, since these activities can be done incrementally. On the other hand, we observe that adding features makes the composition more complex and leads to the so called feature interaction problem.

A feature composed with a base system influences the behavior of the base system. The system $\mathcal{B} \oplus \mathcal{F}_x$ is different from the system $\mathcal{B}$. We then say that the feature $\mathcal{F}_x$ is interacting with the base system $\mathcal{B}$. Several features can be composed together with the system. In that case, features tend to influence the behavior of the system as well as the behavior of each other. We say that the features interact with the system and with each other. Since features interact together, we call this a feature interaction. Section 2.1.2 explains in detail the notion of feature interaction and the problems to which such interactions often lead.

2.1.2 Definition of a Feature Interaction

An interaction between features can be either a *cooperative interaction* or an *interfering interaction*. A *cooperative interaction* occurs when features interact together without causing undesired behaviors or other problems to the system (e.g. deadlocks). If features and their composition are well defined, only *cooperative* interactions should occur. An *interfering interaction* is the contrary of a *cooperative interaction*: it causes incompleteness and inconsistencies in the specification, possibly making it unimplementable. When occurring at run-time, the *interfering* interaction usually harms the system and results in undesired behavior from the user's point of view.

The term *feature interaction* usually refers to an *interfering feature interaction*. In order to simplify the reading, either the acronym *FI* or the expressions *feature interaction* or simply *interaction* are used thereafter to refer to an *interfering feature interaction*.

In a system, different feature interactions can occur for different reasons. We identify two main categories of feature interactions:

Direct interaction – There may be a direct interaction if two or more features are triggered at the same time, by the same event and lead to different, or contradictory results. In this case, the system may behave inconsistently, with non determinism or contradiction (which is a special case of non determinism) in the results.

For example, a user X who subscribes to Call Waiting is notified if someone calls him when he is busy. Suppose that X also subscribes to Voice Mail, a feature that is triggered by an incoming call when he is busy or does not answer after a certain time. If not studied and designed carefully, the composition of these two features can lead to a behavior not desired by the user. For instance, when busy, Voice Mail can be triggered as soon as there is an incoming call. Voice Mail then overrides Call Waiting, which might not be the behavior intended by the user.

Transitive interaction – The transitivity is denoted by the fact that the results of one of the features involved in the composition lead to the triggering of another feature also involved in the composition. An interaction may occur if such transitivity leads to contradictory results between the features or to a loop.

For example, let us suppose that user X subscribes to Call Forward Always and forwards all incoming calls to user Y. Suppose that user Y subscribes to Incoming Call Screening and blocks all incoming calls from user Z. If Z calls X, the Call Forward Always feature forwards Z to Y and expects a connection. This triggers the Incoming Call Screening feature of Y that expects to block the connection. Results are contradictory.

All feature interactions that can emerge from feature compositions are specific to the system specification. Hence, a particular feature interaction that occurs in a given system may not occur in another one. In addition, even if considering the same requirements, different results can be obtained depending on the level of abstraction used to model the features.

As mentioned, an interaction between features can be a *cooperative* or an *interfering* one. Accordingly to our definition of a feature interaction, only interfering interactions are called feature interactions. However, it is important to keep in mind that the difference between *interfering* and *cooperative* interactions can be fuzzy. The borderline is always subjective and depends on the designer or tester interpretation, except if the interaction clearly violates behavioral principles. Given a particular composition of features, the result can be interpreted differently depending on the definition of feature interaction that is being considered.

An example is the following: let us consider that some user X has Call Forward Always to user Y. User Y also subscribes to Call Forward Always but forwards all incoming calls to user Z. Let us imagine that user A calls user X. X forwards him to Y, and, Y forwards him to Z. Then, A ends up ringing Z, while X intended A to ring Y. This can be seen as a *cooperative* feature interaction, since a forward chain is created and the user is *cooperatively* forwarded through it. On the other hand, this can be seen as an *interfering* interaction since the Call Forward feature of Y interferes with the Call Forward feature of X and the final result is different from the the one intended by user X.

2.1.3 Prominence in Telecommunication Systems

Telecommunication systems are all facing feature interactions due to many factors [1]:

- The development of new features is usually stimulated by new technology. New technology often eliminates old obstacles and, by doing so, invalidates deeply established assumptions.

- Most telecommunication devices are designed to be as simple and as standardized as possible. In order to be applicable to these devices, the features developed must also be standardized, sharing a common vocabulary and signal, which is not always the case.

- Telecommunication systems over the world have to inter-operate. Unfortunately, the interfaces allowing them to inter-operate do not always follow standards. This often causes problems when a call is processed through different systems.

- Features in telecommunication systems tend do be exceptions with respect to each other. A certain feature can implement a general behavior while another implements a particular case of this general behavior. Features such as Call Forward Always (general case) and Call Forward on Busy (exception case) illustrate this case.

- The fact that telecommunication systems are aging but are still growing quickly with the addition of hundreds of features tends to be a risk. The more features a system possesses, the more interactions can occur.

- Telecommunication systems have a long tradition of developing projects, and even marketing, by features. Such an organization inevitably discourages thinking across feature boundaries.

- Features are often developed by different teams, especially in case of inter-operating systems such as systems from different vendors or countries. This leads to the fact that features have to deal with other unknown features that cannot always be trusted.

2.2 Existing Approaches

Detecting feature interactions in a product implies to build test cases and to apply them against the product. Removing interactions found in a final product implies internal code modifications, which is usually a high cost task [13].

Avoiding interactions at the design stage of a system does not remove the necessity of testing the final product but considerably reduces the number of tests to build since most of the interactions are already removed or avoided [3, 4]. Hence, approaches that focus on static avoidance and detection of feature interactions at the design level are generally a cost effective option. The approach we present in this thesis follows these principles.

A significant number of approaches, involving related techniques and tools, have been proposed to deal with the feature interaction problem. These approaches have their strengths and limitations and are usually complementary.

We believe that combining them is often better than trying to choose the best one. In this section, we give an overview of different approaches related to our own approach and briefly discuss their strengths and limitations.

Note that this thesis does not address the problem of feature interaction resolution. This is the problem of removing interactions after they are detected. It is a separate problem on which there is abundant literature [3, 4, 9, 10].

2.2.1 Avoidance & Detection Using Patterns

A pattern language is a set of patterns that are used together to solve a problem. This is a method that has been introduced recently. The use of a pattern language provides solutions for the design of features as well as for the resolution or handling of feature interactions.

G. Utas proposes such an approach [14] . This approach uses a pattern language for call processing and for feature interactions. The patterns developed define generic methods to deal with the development of a call model and with interactions (cooperative or interfering) that occur between features.

This approach allows the designer to model the basic system, the features and the interactions between features in a clean way, and thus, to avoid many of feature interactions. However, detection of interactions is not automated and depends on the designer.

2.2.2 Avoidance & Detection Using Use Case Maps

Another possible approach is to use a graphical notation to help designers to get a global view of the control flow of a system. Having a global and simple view of the system and features allows the designer to have a better understanding of the model, and thus, to detect possible feature interactions faster.

Use Case Maps [15, 16] are a scenario based notation. Their purpose is to allow designers to describe the behavior of complex systems in a high-level fashion. Using this notation allows to express requirements using standard constructs.

The notation is intended to be useful for requirements specification, design, testing, maintenance, adaptation, and evolution [17]. Use Case Maps have been used in a number of areas such as requirements engineering and design [18], validation [19], detection and avoidance of undesirable feature interactions [20, 9].

The analysis of the concepts expressed by a UCM system is manual. The tester must visually follow the paths to detect inconsistencies. As for patterns, detection of interactions is not automated and depends on the designer. The possible translation of a UCM model into a formal specification [21] enables the use of an executable model on which automatic validation can be performed.

2.2.3 Detection Using LOTOS

LOTOS (Language of Temporal Ordering Specifications) is a Formal Description Technique (FDT) that has been developed within ISO [6, 8]. It is based on process algebraic methods. Tools allowing the *execution* of a specification allow the application of test scenarios (scenario based validation). Such tools allow the designer to use various techniques to detect feature interactions that can be present in the specification considered.

Goal Oriented Execution is a technique for feature interaction detection that is based on the use of LOTOS tools. LOTOS semantic is defined in terms of axioms and inference rules. Given a behavior, these rules are used to generate the next possible actions as well as the next behavior following the execution of one possible action. A sequence of actions is seen as a trace of actions. *Goal Oriented Execution* uses a type of inference rule that generates traces of actions leading to a preselected action in the specification: the goal.

Applied to the feature interaction problem [22], this technique consists in building a LOTOS specification of the system in consideration together with its features and their properties. Then, identifying goals that correspond to the violation of such properties allows to apply the inference rules of the *Goal Oriented Execution* to check whether such goals can be reached in the specification. If such goals are reachable, a feature interaction exists.

This method however presents some limitations. State explosion [23] can occur due to the number of possible paths required to reach an action. Some actions can possibly remain unreachable. Reaching some goals may imply the use of intermediary goals. In addition, even if all goals are reachable, detecting all feature interactions implies to identify all such sequences of actions corresponding to them and it is not possible to insure that all such sequences have been identified.

2.2.4 Detection Using Permutation Symmetry

Many approaches are based on Finite State Machine (FSM) [24, 25] for the description and specification of features. The methods used for detecting feature interactions in these approaches are often based on exhaustive searches. Such methods are complete and reliable in principle but since they explore the whole space of states and transitions of the system, they usually need a huge computational time and often lead to the so called state explosion problem, even when considering small specifications [26].

Permutation symmetry [26] proposes to reduce the state space by using a state space reduction technique based on the identification and removal of symmetric relations in the specification. This reduction technique keeps the essential relations allowing the detection of the same interactions (in the same number) as in the complete specification with a gain of time.

This method allows a reduction of the number of states and transitions. However, in its current formulation, this method is only applicable to FSM or Petri Nets based specifications and the reduction of the number of states and transitions is not sufficient to eliminate the state explosion problem.

2.2.5 Detection Using Temporal Logic

Temporal logic [27] is a language that provides a set of special operators used to express formal properties in a natural way. This language defines predicates over infinite sequences of states, allowing, for instance, to express the fact that a sequence is *always true*, *sometimes false*, *always false*, etc. Hence, each formula of temporal logic is (in general) satisfied by some sequences of states and falsified by some others.

Such a language can be applied to the feature interaction problem. In paper [28], A.P. Felty and K.S. Namjoshi present an approach that consists in specifying features using temporal logic. They show that inconsistencies between features can be automatically identified using existing model checking tools [29].

After experiments on a set of feature specifications derived from Telcordia standards, the approach revealed itself as being *reasonably efficient and quite accurate* [28]. Most of the interactions given in Telcordia standards were detected, as well as new ones. However, as mentioned by the authors, the logics used in this work are limited to next-state descriptions, so no liveness properties can be expressed.

2.2.6 Static Detection Method Without State Enumeration

T. Ohta and T. Yoneda developed an approach based on a state transition representation of features and knowledge *elicitation* [30, 31]. Features are represented using a formal language. The authors developed an algorithm that *elicits* knowledge from the services specification. This knowledge is used in conjunction with rules to detect feature interactions.

This approach allows to avoid problems such as state explosion, high computational time and low coverage. However, the feature interaction detection remains manual. This approach is explained in greater detail in chapter 6, section 6.4.

2.2.7 Related Problems

In spite of the detection methods presented, some problems remain:

- Methods presented do not identify all interactions. Even with a strong and precise language to represent features and with tools to automate the detection of possible feature interactions at the design stage, many interactions can remain undetected.

- Adding new features to a system implies testing their interactions with the existing ones. It also implies considering more possible paths and states. This can lead to state explosion problems, making the integration testing difficult, if not impossible.

Today's environment in telecommunications is competitive. The number of features increases very quickly, leading to an important number of potential interactions: detecting feature interactions is necessary, but also time consuming. The project on which our work is based gave us the opportunity to propose a method that allows the detection of many types of feature interactions in a reasonable time.

2.3 Structure of the Project

Our work is related to a collaborative Research and Development project involving the University of Ottawa *(Telecommunications Software Engineering Research Group, School of Information Technology and Engineering)*, Mitel Corp. and Communication and Information Technology Ontario (CITO). The project has a wide scope and involves the use of many

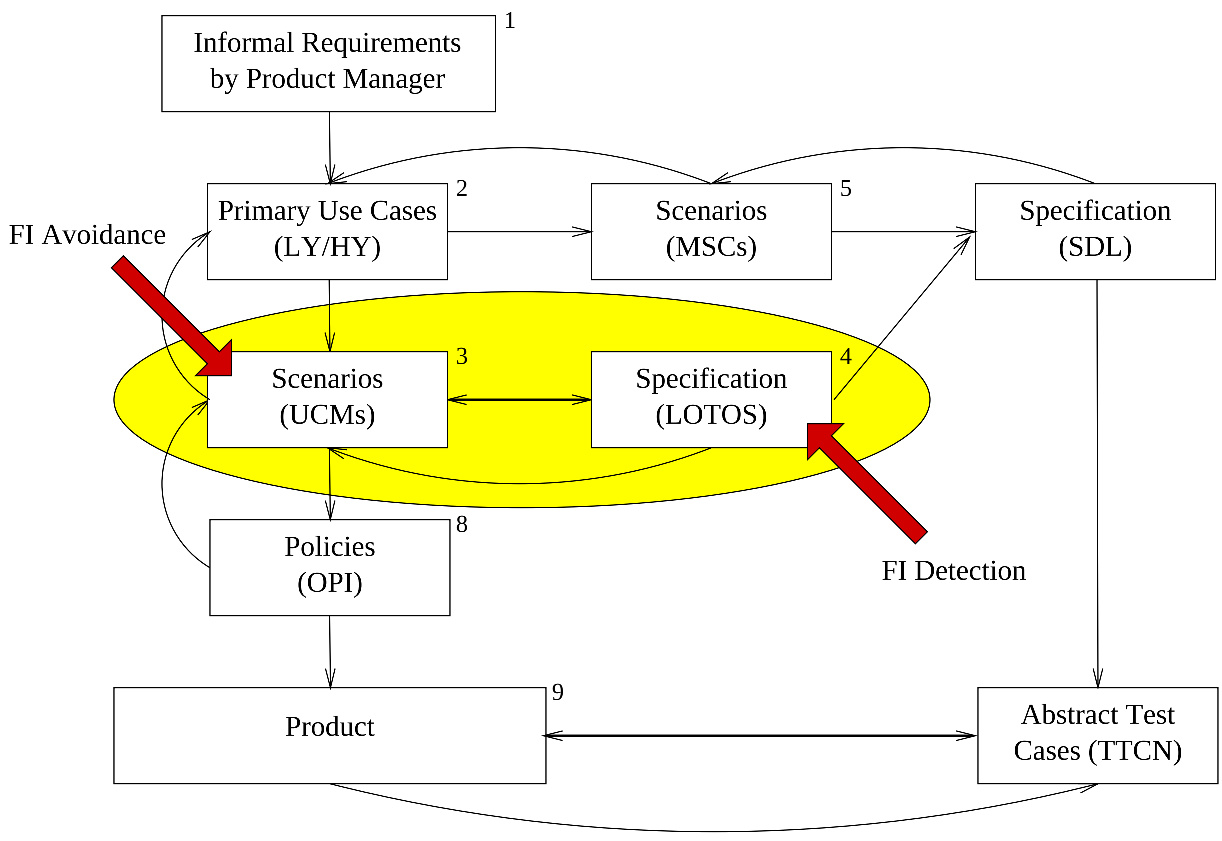

Figure 2.1: FI Detection in a Formal Product Development Model

techniques, some (e.g. UCM, LOTOS) discussed in this thesis, and others (e.g. SDL, OPI, TTCN) not discussed here. In order to give a general description of the project, we are obliged to mention below a number of concepts on which we do not elaborate because they are not used in the rest of the thesis. The reader who is not familiar with these concepts can ignore this section without harm.

Mitel Corp. provided us with a set of Use Case Maps (about 110 maps) representing an agent-based telephony system with 9 features. This system and the features it contains are presented in chapter 3, section 3.2. The aim of the project is to establish a Software Engineering process for the fast specification and testing of such a system.

As illustrated in fig. 2.1, the method starts with informal requirements (1) from which primary use cases (2) are derived. The project splits in two iterative processes:

- One (5) consists in building scenarios using Message Sequence Charts (MSC) [32]. These scenarios are used to produce and validate the SDL (6) [7, 8] specification. The SDL specification allows the derivation of Abstract Test Cases (TTCN) (7) [33] used to test the final product (9).

- The other one consists in building scenarios using Use Case Maps (3). These scenarios are used to provide OPI policies [34] (8), used in the implementation of the final product. In parallel, they are used to produce a LOTOS specification (4) which provides an executable semantic to UCM. The LOTOS specification is validated using scenarios obtained from UCM, providing testing information for the SDL model.

2.3.1 Initial Approach

As mentioned in section 2.2, our methodology focuses on the static detection and validation of feature interactions in the early stages of development of a system. Our initial approach is part of boxes (3) and (4) in fig. 2.1 and is composed of the following steps:

Use Case Maps Model

The methodology starts by employing Use Case Maps (UCM) to build a general description of the system (3). As explained in section 2.2.2, UCM are a scenario based notation. Their purpose is to allow designers to describe the behavior of complex systems in a high-level fashion. Using this notation allows to express requirements using standard constructs and provides designers, product managers, and even non-specialists with a graphical representation that offers a global view of a system.

The UCM validation targets *FI avoidance*. This graphical representation enables designers to focus on the *big picture* and, we believe, to avoid and detect design defects at early stages. Because it is high-level, this notation is implementation-independent. Therefore, validated UCM are reusable on different platforms and different products, and at different stages of product development and use.

We see UCM as being efficient as an intermediate representation between requirements and formal specification. The UCM concepts and notation are presented in detail in chapter 3, section 3.1. In order to validate the design and detect possible errors, the UCM representation of the system is implemented using the Formal Description Technique (FDT) LOTOS and its tools.

LOTOS Specification

Introduced in section 2.2.3, LOTOS (Language of Temporal Ordering Specifications) is a Formal Description Technique (FDT). The use of LOTOS allows to build a formal specification of a system. Verification and validation are performed on this model using LOTOS tools. LOTOS concepts are presented in chapter 3, section 3.3.1.

The derivation of the LOTOS specification is currently done by hand, although a partial automation is being considered. Required knowledge here is a good understanding of the UCM system, the features being considered, expertise with UCM, LOTOS, and the techniques used to translate UCM into LOTOS.

To derive the LOTOS specification, the UCM model is analyzed to determine an appropriate specification structure. During this analysis, the designer makes choices concerning the mapping of the UCM model structure and constructs into LOTOS. Since Use Case Maps are used at a higher level of abstraction than LOTOS, this analysis gives the opportunity of inspecting the documentation and detecting missing parts, contradictions, and incoherences. The translation guidelines are presented in chapter 3, section 3.3.2.

Feature Interaction Detection

The LOTOS validation targets *FI detection* based on scenario-based testing. Scenarios are derived from UCM for validation and verification purposes against the LOTOS specification. Note that these scenarios can be re-used later on for similar activities against an SDL specification of the system, up to the generation of test cases in machine-readable format.

The derivation of scenarios from the UCM model is done by hand. Again, required knowledge here is expertise with UCM, LOTOS, and the features being considered. Once translated into LOTOS, test scenarios are used to validate the LOTOS specification, then to test the system. Scenarios may be categorized according to what they are meant to test: basic system properties, individual features properties or feature interactions.

The testing is performed using LOLA *(LOtos LAboratory, Universidad Politécnica de Madrid)* [35], a free-ware tool. Each test scenario, translated into LOTOS, is synchronized with the LOTOS specification (see chapter 3, section 3.3.1 for explanations on synchronization). LOLA reports whether visiting the composed behavior leads to success for all execution branches (MUST PASS), for only some branches (MAY PASS) or for none of them (REJECT). Scenario based validation is explained in chapter 5, section 5.1.

2.3.2 Refinements

In this approach, the derivation of test scenarios for testing feature interactions is done by hand. Producing scenarios manually is not a long task if the number of features is small. However, a large number of features makes it almost impossible, especially because the number of possible combinations increases considerably with the addition of new features due to the number of new possible combinations it creates.

A first refinement of this process is to automate the identification of possible feature interactions in requirements as well as their detection in a specification. This is the subject of this thesis. Briefly, such an approach requires two steps. The first step takes place after the users' requirements phase. It is based on feature definitions only and consists in filtering the incoherences corresponding to potential interactions. The next step takes place after the specification phase and consists of the automatic derivation of validation test suites for each and every incoherence identified in the first step. These test suites are composed of scenarios derived using information relative to the structure and syntax of the specification. They are applied to the specification to detect feature interactions.

This refinement increases the strength of the current approach. It allows the avoidance and detection of feature interactions on the basis of a simple and fast description of the features. Moreover, the automatic derivation of test suites for the validation of the specification considerably reduces the time of the testing process. This process is explained in section 2.4.

2.4 The Proposed Filtering and Validation Process

We should, at this point, provide a general overview of the filtering of incoherences and feature interactions detection process. We provide a few informal definitions in order to enable a first understanding of what follows.

Filtering considers the feature definitions given in requirements and analyzes these definitions in order to identify the presence of incoherences that can lead to potential interactions in the specification. The identification of such incoherences is seen as a filter since a number of potential interactions can be identified at the requirements level. These incoherences and the rules used for their identification are presented in chapter 4.

In this thesis, the word *incoherence* refers to incoherences that are present between features at the requirements level and that correspond to potential interactions possibly present in the specification. The words *interaction* and *feature interaction* refer to a feature interaction present (or not) in the specification.

As illustrated in fig. 2.2, the process starts with requirements and splits into two paths. The paths are in parallel but they depend on each other. The one on the left concerns the identification of incoherences and derivation of test suites to detect feature interactions in the specification. The one on the right illustrates the process of deriving of the Use Case Maps model, later translated into a LOTOS specification. The whole process is composed of the following steps:

Incoherences Filtering (1)

Even if informal, requirements can provide enough information for the identification of incoherences between two features.

The information about features is extracted from the requirements and mapped into a Prolog [36, 37] representation (see chapter 5, section 5.2 for explanations about Prolog). The descriptions of features are analyzed by our tool which combines them by pairs and identifies the incoherences they present. As previously mentioned in section 2.4, these incoherences, automatically identified, reveal interactions that can possibly occur in the specification.

At this time in the design process, the complete detailed specification of the features is not known, and thus, the analysis of incoherences is only based on the information that is extracted from the requirements. Due to this, it is not possible to assert that the incoherences identified lead to feature interactions in the specification, nor that all absence of incoherences implies the absence of feature interactions.

Even if not absolute, these results provide useful information regarding the features that, once combined together, can lead to interactions. This information can be used by designers

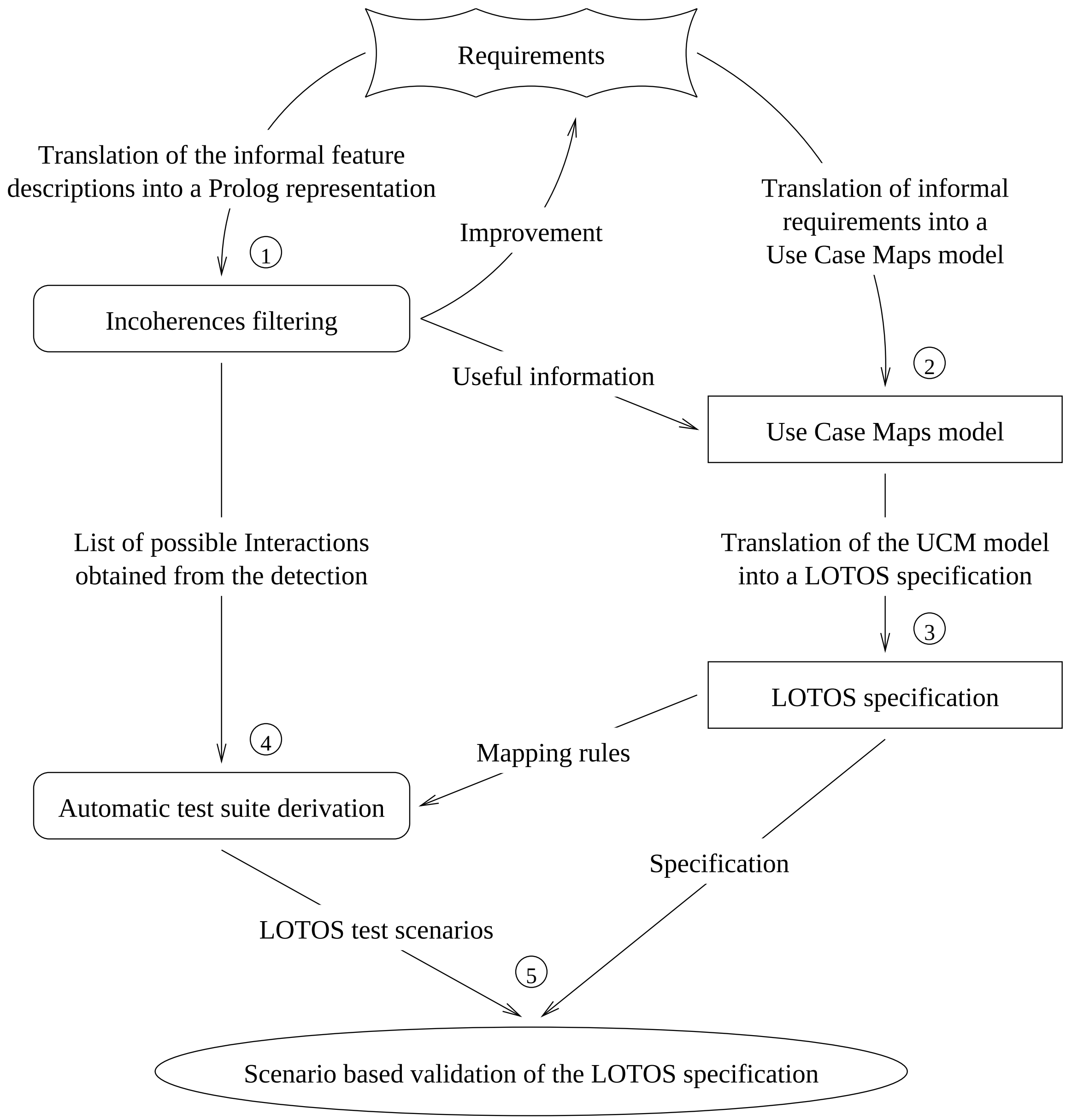

Figure 2.2: Feature Interaction Validation Process

to refine requirements and to avoid the presence of such interactions in the specification. The filtering theory and its automation are respectively presented in chapter 4 and chapter 5.

Use Case Maps Model (2)

Requirements are translated in a UCM model in the same way it is presented in section 2.3.1, and explained in detail in chapter 3, section 3.1. It is built in accordance with the knowledge about incoherences obtained from the filtering process. Thus, the identification of incoherences helps to avoid their presence in the UCM model.

LOTOS Specification (3)

The LOTOS specification is done by manually translating the Use Case Maps into LOTOS, as it is presented in section 2.3.1 and explained in detail in chapter 3, section 3.3.2.

Automatic Test Suites Derivation (4)

Since the LOTOS model is executable, test scenarios can be run against it. Deriving test scenarios corresponding to the incoherences identified allows to verify whether or not the interactions they correspond to exist in the specification.

The structure and syntax of scenarios rely on the structure and syntax of the specification to which they are applied. The incoherences are represented as scenarios containing actions. These actions are translated into LOTOS test scenarios via mapping rules. These rules are built with respect to the structure and syntax of the LOTOS specification. Their purpose is to provide a correspondence between the Prolog representation of a feature and its implementation in the LOTOS model.

Using LOTOS testing tools, the test suites are applied against the specification to check whether or not interactions are present. This process is not strictly dependent on the use of LOTOS; other specification languages could be used. The automatic derivation of test suites is presented in chapter 5, section 5.3.

Scenario Based Validation of the LOTOS Specification (5)

The testing of feature interactions in the LOTOS specification is done by applying scenarios against the specification using LOLA, as previously presented in section 2.3.1 and explained in detail in chapter 5, section 5.1.

2.5 In Summary

We have introduced the framework on which our thesis is based. We have presented our definitions of a feature and a feature interaction and enumerated some of the current approaches used to deal with feature interactions. We have proposed an approach combining a known approach and a filtering process for the identification of incoherences and detection of feature interactions in a model. This approach is presented in detail in the rest of the thesis.

Chapter 3

Use Case Maps, LOTOS & Mitel's System

This chapter introduces two techniques used in the project, UCM and LOTOS, and introduces the system designed using these two techniques. We present Use Case Maps, their purpose and the way they can be used to specify systems. We present the system provided by Mitel Corp. as a set of UCM and explain how the translation of this model into a LOTOS specification is done and which considerations and decisions must be made.

3.1 Presentation of Use Case Maps

The task of designing and describing complex systems can be very laborious, even for experienced analysts. It is a good option to describe such systems with a high level of abstraction and in an efficient way such that they are fast to design and to understand.

As mentioned in chapter 2, section 2.2.2, Use Case Maps (UCM) are an adequate technique for representing complex systems. They are an informal design notation; their semantic is not precisely specified: it is intended to permit different interpretations. UCM allow the reader to have a global idea of the system at a glance. In addition, a tool, the *Use Case Maps Navigator* [38, 17], assists the designers in the design and the syntactic correctness of the UCM model they develop.

This section gives an overview of the UCM with respect to the elements of the notation used in the design of the Mitel system. A few additional elements are presented in order to give to the reader a general understanding. More complete dictionaries of the notation can be found in [15, 17, 16].

3.1.1 Basic Notation & Interpretation

The basic notation is composed of components, scenario paths and responsibilities. Fig 3.1 shows an example of a simple system represented using UCM.

Components – They are used to specify the structure of the system. A system can contain several components and a component can contain several sub-components. Different shapes can be used to represent several kinds of objects depending on what the designer wants to express.

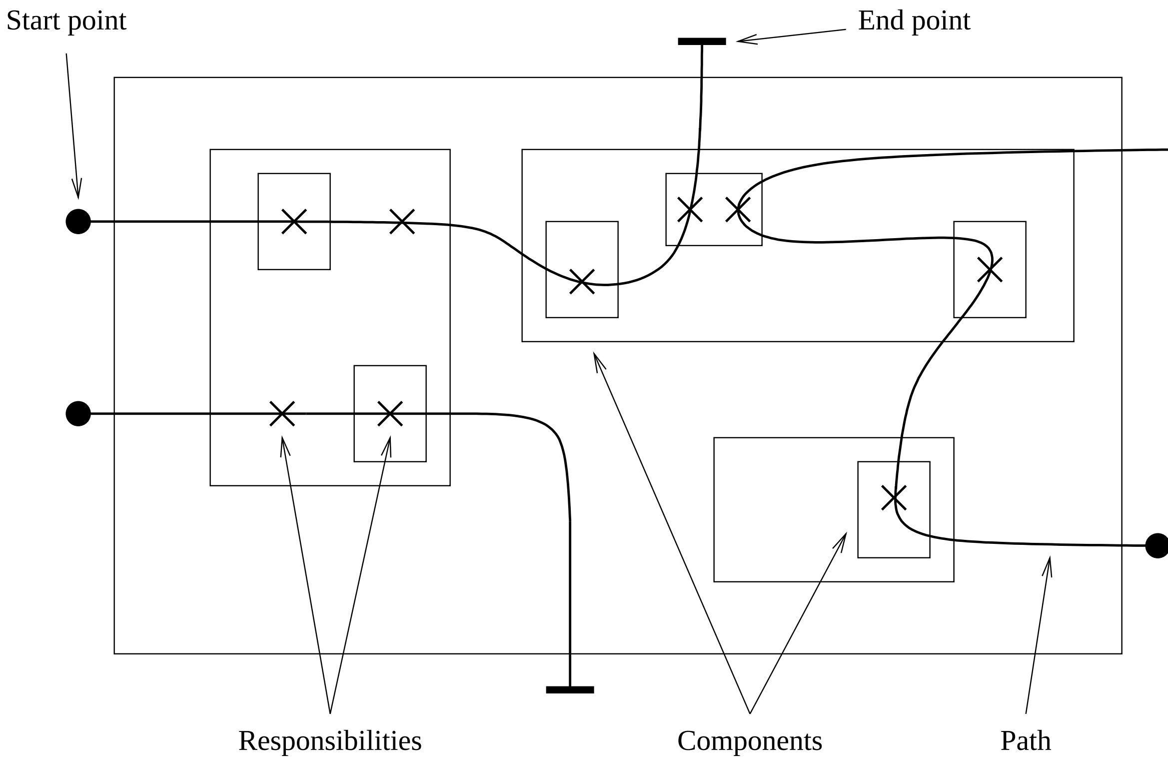

Figure 3.1: Use Case Maps Basic Notation

Scenario paths – Also called routes, they are represented by curved lines and are used to express the behavior of the scenario along time. A path starts with a start point, represented as a bullet, and terminates with an end point, represented as a bar. Paths navigate across the components to mark the places where responsibilities take place.

Responsibility points – Responsibilities are represented by crosses and are situated on paths. A responsibility is used to represent some action that takes place during the scenario. It can represent different types of actions depending on the system designed. For instance, in case of a telephony system, a responsibility can be used to represent an off-hook event.

3.1.2 Scenario Paths Notation

The scenario paths notation contains all that is needed to express scenarios using paths. This include interactions between paths, shared routes and concurrent routes. This section explains the elements of the notation. In addition, we show some variations that can be obtained by combining elements of the notation together.

Shared Routes

Shared routes represent segments of paths that can be shared by two paths. They can be used to express the fact that scenarios have common behavior parts. They are also used to express the possibility of a choice between two routes. Fig. 3.2 shows the operators used.

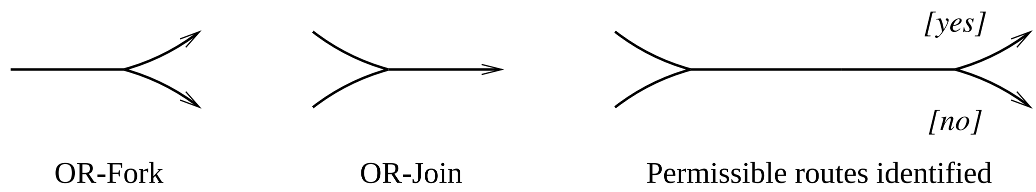

Figure 3.2: Shared Routes

OR-Fork – The OR-Fork expresses a choice. It represents the exclusive `OR` operation on a path divided into two possible routes. Only one of the two possible routes is followed.

OR-Join – The OR-Join expresses the fact that two different possible paths of a scenario join and share the same segment of path. Note that this does not express parallelism. In that case, only one of the two paths is active. The fact that the paths share the same segment means that, whatever path is active, the same actions take place.

Permissible routes identified – To indicate which path must be followed, alternatives may be identified by labels or by conditions associated to the corresponding sub-path.

Concurrent Routes

Concurrent routes are used to express parallelism. They allow someone to indicate a point where a scenario splits into two (or more) different branches that continue their execution independently. They also allow someone to indicate a point where branches need to merge. Fig. 3.3 presents the operators used.

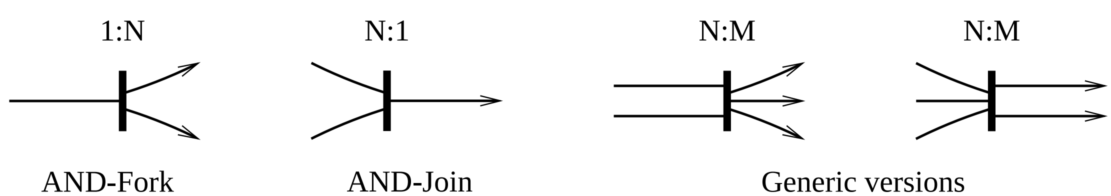

Figure 3.3: Concurrent Routes

AND-Fork – The AND-Fork splits one path into two paths that proceed in parallel. Paths are fully independent from each other. Thus, this operator expresses parallelism and can be used to model concurrent scenarios.

AND-Join – AND-Join indicates that two paths join each other to continue together (as being the same one). This operator can be used to model concurrent scenarios that need to do common actions at the same time.

Generic versions – As shown by fig. 3.3, AND-Fork relationships are usually 1 to N and AND-Joins relationships are usually N to 1. However, it is possible to use a generic version of them to express the fact that N branches are splitting into M ones, or to express the fact that N branches are joining to become M ones.

Variations on AND-Forks/Joins

Composing the Fork and Join operators is possible and can be used to express temporary concurrency, synchronization or rendezvous, as illustrated by fig. 3.4.

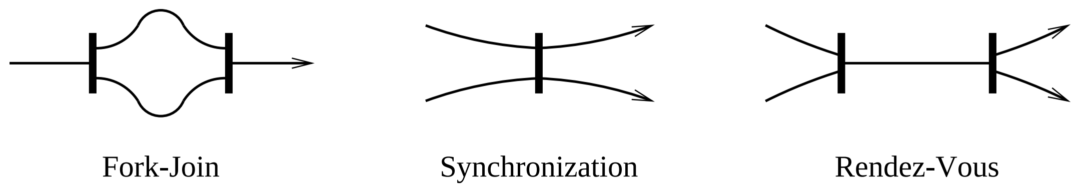

Figure 3.4: Variations on AND-Forks/Joins

Fork-Join – Fork-Join is a composed operator using an AND-Fork followed by an AND-Join. It expresses the fact that a path splits into two parallel sub-paths that join later on and can be used to express temporary parallel behaviors.

Synchronization – Synchronization is used to denote that, at a certain point, paths must synchronize. Two paths are able to pass the synchronization point only once they have both arrived. A generic version of this operator can be used to make more than two paths synchronize.

Rendezvous – Rendezvous is a composed operator using an AND-Join followed by an AND-Fork. It expresses the fact that two parallel paths are joining and synchronizing along the same one for a time, and then fork again. This can be used to express temporary common behaviors of synchronization of scenarios.

Paths Interactions

Path interactions are used to represent two (or more) routes interacting together. We distinguish two kinds of interactions: synchronous and asynchronous, as illustrated by fig. 3.5.

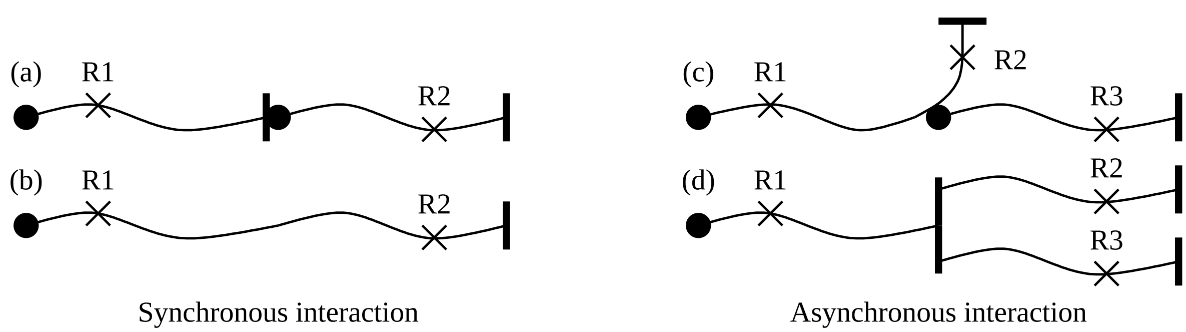

Figure 3.5: Paths Interactions

Synchronous interaction – The synchronous interaction between two paths is represented by fig. 3.5-(a) and shows that reaching the end of the first path triggers the second one. The effect is similar of one longer path with the constituent segments joined end to end, represented by fig. 3.5-`(b)`.

Asynchronous interaction – The asynchronous interaction, represented by fig. 3.5-`(c)` represents the fact that, the second path is triggered by the first one and that the first one continues its execution. The effect is similar to one path splitting in two concurrent segments, as shown by fig. 3.5-`(d)`.

3.1.3 Components Notation

Representing components in Use Case Maps can be done using different shapes depending on the object that has to be represented. A component can be static or dynamic. Static components are parts of the system that do not change. Dynamic components are components that can be dynamically created, moved or destroyed. A component can have different attributes. The shape of a component illustrates its attributes. This section introduces the different components and their corresponding representation, as well as the representation of their attributes.

Component Types

Fig. 3.6 illustrates the different shapes that can be used to model static and dynamic components. Each shape corresponds to a particular kind of component.

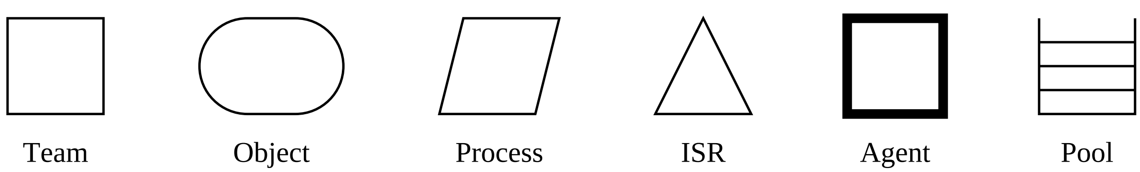

Figure 3.6: Component Types

Team – The team component is a generic container. It may contain components of any types. It is a sort of default component that is generically used in Use Case Maps models as illustrated by fig. 3.1.

Object – Objects represent passive components. They are used to represent parts of the system that are not executing. Such objects act as data and can represent specific components such as databases or more abstract concepts such as a call in a telephony system.

Process – While objects represent passive components, processes represent components that are active. Such components represent executing parts of a system, a computation for instance. Processes also permit the representation of distributed systems and their execution.

ISR – ISR, Interrupt Service Request is used to model the requests of service interruption that occur in the system.

Agent – Agents are used to represent agents of a system.

Pool – Pools are containers for Dynamic Components that are not executing. Since such components are considered as data, a pool can be used to represent data storage.

Component Attributes

Different types of components exist and can have different attributes depending on their functionality. Shapes presented in fig. 3.7 are not bounded to the team component. They are generic and illustrate the possible attributes for any component.

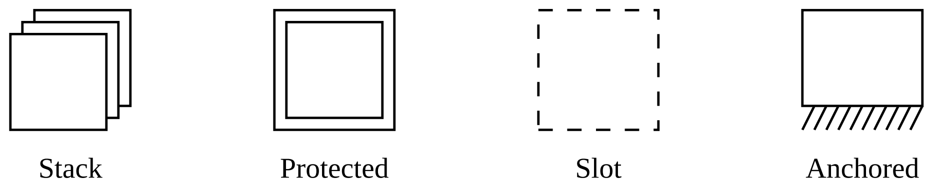

Figure 3.7: Component Attributes

Stack – Stacks are used to represent multiple instances of an object. For example, a stack of agents can represent multiple instances of the same kind of agent. Note that these are not stacks in the sense of first-in-last-out data structures.

Protected – The attribute is used to express mutual exclusion on a component. Using it permits the modeling of mutual exclusion concepts such as semaphores and monitors.

Slot – A slot is a placeholder for dynamic components as operational units. Slots may be populated with different instances of different components at different times.

Anchored – Anchored components are found in plug-ins. A component that is anchored refers to a component that is defined in another map. This can be used to model the fact of exporting components.

3.1.4 Additional Notations

Use Case Maps provide additional notation for extending the modeling possibility: stubs and plug-ins, timers, aborts, grounds and shared responsibilities. These are presented below.

Stubs & Plug-Ins

Stubs can be seen as modules of functions. They are containers for sub-maps (called plug-ins). Using stubs makes it possible to abstract from parts of the system by encapsulating them. Moreover, stubs can be used to decompose a system into different levels, permitting more modularity.

Two kinds of stubs are identified: static stubs and dynamic stubs. Fig 3.8 shows an example of a static and a dynamic stub with their plug-ins.

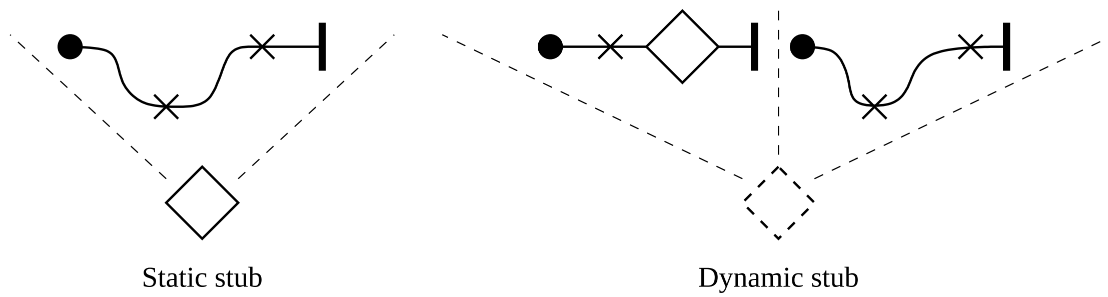

Figure 3.8: Static & Dynamic Stubs

Static Stub – Static Stubs are represented as solid diamonds. They contain one sub-map (plug-in) and are mainly used for the decomposition of system into sub parts.

Dynamic Stub – Dynamic stubs, represented as dashed diamonds, can contain several plug-ins. When entering a dynamic stub, the selection of the plug-in to use can be determined by using selection policies usually described with pre-conditions. It is possible to select several plug-ins at the same time, for sequential or parallel composition, but adequate detailed documentation needs to be provided outside of the UCM.

Timers, Aborts, Grounds & Shared Responsibilities

The elements illustrated in fig. 3.9 are used to indicate possible failures, aborts, timeouts and shared responsibilities.

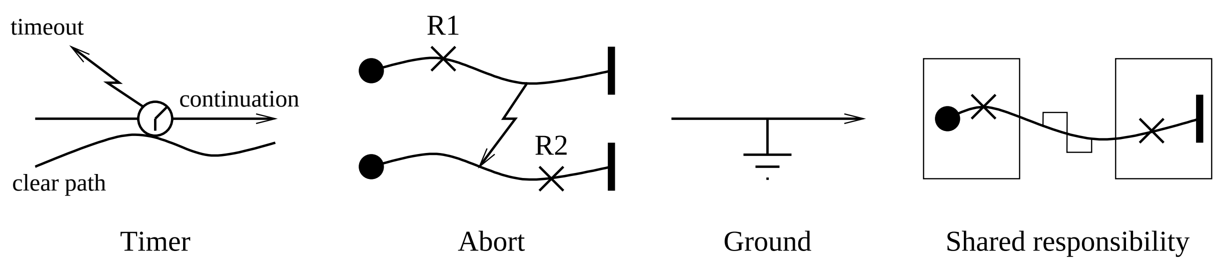

Figure 3.9: Timers, Aborts, Grounds & Shared Responsibilities

Timer – A timer is a special waiting place. It is associated with a continuation, a timeout and a clear path. If the timer is reached before it times out, the continuation path is followed. If the timer is not reached and a timeout occurs, the timeout path is the one followed. In any case, the clear path can be used to reset the timer.

Abort – The abort operator expresses the fact that one path may abort another. Fig. 3.9 shows us an example where the top paths aborts the bottom one after the execution of responsibility $R1$.

Ground – The ground element indicates a potential failure point along a path.

Shared responsibility – Shared responsibility represent a complex activity that involves negotiation between two or more components.

3.2 Mitel System

This section presents an overview of the structure of the system on which our work is based, as well as a description of the features integrated to this system.

3.2.1 Use Case Maps Model

The Mitel system architecture, including its various features, was captured using UCM, for which an example is shown in fig. 3.10. This figure shows part of the Root UCM, limited to the originating DA, CA and LA elements.

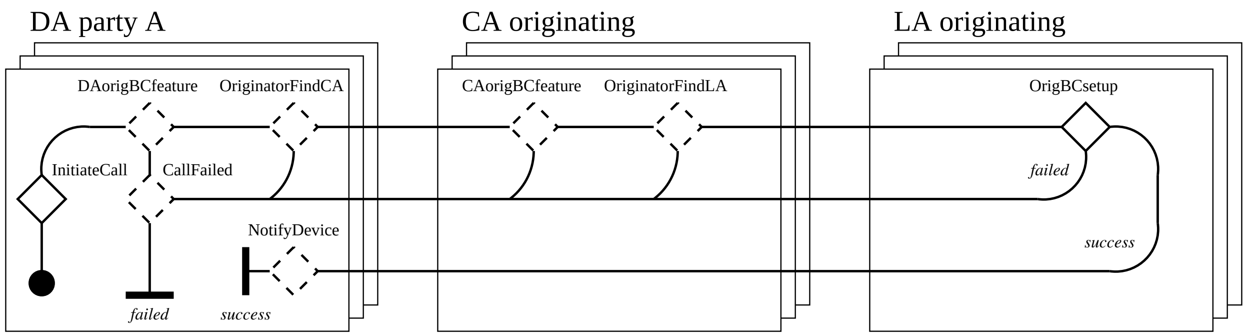

Figure 3.10: Part of the Basic Call Root UCM

Logical Agent (LA) – A LA represents a logical endpoint of the call in a switch. A phone number is associated with a LA. A LA is often associated with the location.

Communicating Agent (CA) – A CA represents a user. It knows about and deals with restrictions and privileges given to a specific user. A CA can be seen as associated to the *role* of a user. A given user can be associated with different CAs.

Device Agent (DA) – A DA represents the physical endpoint of a call. This can be an actual telephone, a computer capable of Voice over IP or some other device. A user can have several DAs depending on the number of communication devices he possesses.

Call Object (CO) – A CO is an intermediary between two LAs participating in a call.

Let us consider a company where employees wear identification badges. These badges broadcast their location to the phone system of the company. The phone system treats incoming phone calls in the following way: as soon as a call comes into the phone system for a particular employee, the system locates the employee and rings the nearest phone.

Let us imagine that a call enters a system. This call enters through a DA which represents the trunk on which the call is coming in. That DA then routes the call to a CA which represents an outside user of the system. The CA involved then chooses a LA built for an outside user. The chosen LA creates the Call Object which can find the LA representing the number being called. The call goes to that LA. The LA finds a CA which represents the user corresponding to the number that was dialed. The CA then locates the user in the building.

Using the information it has about the user's location and the locations of the phones in the building, it chooses a DA. The DA then rings the phone it represents (the nearest available phone).

As mentioned, our case study is based on the Use Case Maps model of an agent system given by Mitel Corp. The Use Case Maps model is composed of 110 different maps distributed in 7 different levels of sub-maps. The UCM were structured in a hierarchical way to describe the 9 features that were considered in this project. Nearly 60 stubs were used along the way, and many plug-in UCM were reused in more than one stub. The responsibilities are found in the plug-ins, sometimes several levels deep in the hierarchy.

Due to the already large number of stubs and complex functionality in the current UCM, and given the reusability of these, we expect to need few additional UCM to capture extra features. This section presents the structure of Use Case Maps model representing the system and its components.

3.2.2 Features Considered

The use of UCM can help directly in the detection and avoidance of many undesirable actions before any prototype is generated. By inspecting specific locations such as dynamic stubs, designers may select appropriate strategies. For instance, a selection policy where preconditions are incomplete or overlapping (non-deterministic) is likely to cause problems. The detection and resolution of many problems can hence be done locally at the stub level, which is, in our experience, far better than doing it at the system level. The correctness is further validated when testing the LOTOS specification. The features considered are described below:

Call Forward Always – Party A calls party B. Party A ends up ringing party C, as a result of party B's request.

Call Forward On Busy – Party A calls party B. Party B is busy. Party A ends up ringing party C, as a result of party B's request.

Auto-Recall – Party A calls party B who is busy. When party B cease being busy, party B is notified of the auto-recall. When party B responds to the notification, a call is made from party B to party A.

Call Waiting – Party A is talking to Party C. Party B calls Party A. Party A receives a notification that party B is on the line. Party A may then switch between talking to party C and party B.

Timed Reminder – Party A sets a timed reminder for a specific time. At the previously set time, party A's phone rings and when party A answers, a message is played.

OutgoingCall Screening – Party A is not permitted to call party B.

Call Hold – Parties A and B are connected. Party B presses a *Hold* button. Party A ends up *on hold*. Party B may retrieve party A at any time.

Call Pickup – This feature allows a party to use his phone and dial a code to answer a call made to another party.

Call Transfer (unsupervised) – Party C is connected to party A. A presses the *Transfer* button, dials B's number, and C ends up ringing B.

3.3 From Use Case Maps to LOTOS

The translation of a UCM model into a LOTOS [19, 21] specification corresponds to the formalization and mapping of an abstract and informal model into a more specific one, while remaining at the design level. In order to do this mapping, the UCM system must be analyzed such that any missing information, confusion or error is detected and corrected.

3.3.1 LOTOS Concepts

LOTOS [6] represents the behavior of a system by using *actions* and *behavior expressions*. Actions represent basic behaviors of the system, for example, *offhook* or *ring*. There is also an internal (invisible) action written as **i**. Three basic behavior expressions are **stop** (also called *deadlock* or *unsuccessful termination*), **exit** (successful termination) and process instantiation P[G](V), where P is the name of a LOTOS process, G is a set of gate parameters (see below for the concept of gates), and V is a set of value parameters. Given behavior expressions B, B1, B2, etc. and actions a, a1, a2, etc., LOTOS operators can be used to construct more complex behavior expressions as it is shown below.

a; B – The *action prefix* operator **;** means that the system offers an action a followed by behavior B.

B1 [] **B2** – The *choice* operator [] means that the next action offer can be obtained either from B1 or from B2. The other behavior expression is discarded.

B1 || **B2** – The *full synchronization* parallel operator || means that a common next action from behavior expressions B1 and B2 has to be found in order for the system to proceed. If such actions exist, they are offered (synchronization) and then the next common action is obtained and so on.

B1 ||| **B2** – The *interleaving* operator ||| means that at any point, independent actions from behavior expression B1 or B2 can be offered.

B1 |[**a1, a2, ..., an**]| **B2** – The *selective synchronization* operator |[]| is a generalization of the full synchronization and interleave operators. It means that on actions a1, a2, ...an, B1 and B2 must synchronize; on other actions they interleave.

B1 [> **B2** – The *disable* operator [> means that at any time during the execution of B1, B2 can take over, thus terminating B1.

B1 >> **B2** – The *enable* operator >> means that, provided that B1 has completed successfully (**exit** behavior), B2 can start.

hide a in B – this internalizes (transforms to **i**) all offerings of actions a in B.

LOTOS includes also 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 !subscriber ?destination:destination_sort
```

denotes an action on gate *switch* where the current value of the value identifier subscriber is offered, and a value for *destination* is queried simultaneously. Offers and queries are called *experiments*. Note that when no data is involved, actions are simply gate names. With data, actions are gate names with data offers and queries.

A basic concept in LOTOS is the *expansion*. Any LOTOS behavior expression can be rewritten as an equivalent expression containing only choice, action prefix, **stop** and **exit** (although this expression can be infinite). An expanded LOTOS specification represents directly the *labelled transition system* (LTS) of the system in consideration (an LTS is like a Finite-State Machine, having states and transitions labelled by actions denoting synchronization events). Each alternative path in an expanded specification, or each branch in an LTS, explicitly represents one possible sequence of actions in the system. Sequences of actions are called *traces*. Traces must include only visible actions, however invisible actions often are also shown, for completeness.

LOTOS has two main assets in the area of specifying telephony systems and their features: it is capable of representing clearly system structures, and it has a good set of validation tools. Concerning the representation of telephony system structures, LOTOS operators allow us to represent clearly agents that coexist independently (interleave), communicate (selective synchronization), follow each other's actions (enable), or can interrupt each other (disable). Internal system actions can be hidden. Several *specification styles* are possible in LOTOS. Some styles are appropriate for representing abstractly system requirements, while others are appropriate for representing implementation structures. Let us consider the two following processes DA and CA:

```
process DA[ DE_to_CE,  CE_to_DE ]: exit:=
  DE_to_CE; exit
endproc

process CA[ CE_to_DE, DE_to_CE ]: exit:=
  DE_to_CE; exit
endproc
```

These can be combined to require synchronization over all their gates as follows:

```
DA[ DE_to_CE, CE_to_DE ]
  ||
CA[ CE_to_DE, DE_to_CE ]
```

The synchronizing points, or *rendezvous* points, between processes DA and CA are then all actions that involve the gates DE_to_CE and CE_to_DE. If one of the processes wants to fire an action with gate DE_to_CE or CE_to_DE, the other process must also be willing to do accordingly for the action to become executable. The two processes then synchronize or *get together* to execute this common action. Synchronizing on an action also implies an agreement on the values of the parameters of the action, namely the experiments. It is possible to accept *(query)* a value, represented by **GATE ?valueId:sort**, or to offer *(shriek)* one, represented by **GATE !value**. In all cases involving synchronization, there must be a match in the number of experiments, their order, their sorts and their values. Let us consider two processes P1 and P2 synchronizing on gate COM with an experiment of sort Natural. The four following situations can occur:

1. P1 = **COM !1** and P2 = **COM ?y:Natural**
   P1 offers **1** to P2, which binds it to the value identifier **y** of sort Natural.

2. P1 = **COM !1** and P2 = **COM !1**
   P1 and P2 agree on the value 1.

3. P1 = **COM !1** and P2 = **COM !2**
   P1 and P2 do ***not*** agree on the value and synchronization does ***not*** occur.

4. P1 = **COM ?x:Natural** and P2 = **COM ?y:Natural**
   A value of sort Natural is determined non-deterministically (e.g. provided by the environment) and is bound to both **x** and **y**.

Combined with the interleaving operator (|||), experiments may also be used to provide a more selective kind of synchronization. This subtlety is called *gate splitting*. A typical application is a switch that may wish to synchronize with any phone, but one at a time. The phone processes are interleaved while the switch is ready to synchronize with any of them:

```
switch[ COM ]
  |[ COM ]|
(
  phone[ COM ](p_1) ||| phone[ COM ](p_2) ||| ... ||| phone[ COM ](p_n)
)
```

Then, an experiment is included in an action to uniquely identify each phone such that the switch may selectively synchronize with a specific one. To query a telephone number from the process phone p_1, the switch can specify the action:

```
COM !p_1 !dialTone ?telephone:number
```

Let us consider that a DA must communicate with two different CAs. In a simplistic approach, one can consider a set of gates (one for each direction) for the communication with each of the two CAs. This produces an undesirable static structure: notice that the set of gates of a DA has to be changed whenever a CA is added or removed. On the other hand, *gate splitting* makes it possible to have more dynamic structures. The previous definitions of DA and CA can be extended with gate splitting to a system containing three processes (e.g. two CAs and one DA). CA processes are instantiated with a parameter that predefines their identity. The process definitions become:

```
process DA[ DE_to_CE, CE_to_DE ]: exit:=
  DE_to_CE !cA_a !connect; exit
   []
  DE_to_CE !cA_b !disconnect; exit
endproc

process CA[ CE_to_DE, DE_to_CE ](cA:CAID):exit :=
  DE_to_CE !cA ?message:Data; exit
endproc
```

And the system behavior is the following :

```
DA[ DE_to_CE, CE_to_DE ]
  |[ DE_to_CE, CE_to_DE ]|
(
  CA[ CE_to_DE, DE_to_CE ](cA_a of CAID)
   |||
  CA[ CE_to_DE, DE_to_CE ](cA_b of CAID)
)
```

Each instance of CA shrieks its own identifier (cA_a and cA_b, respectively) and queries a message of sort *Data*, on the same gate DE_to_CE. The DA may either synchronize with CA cA_a through the event DE_to_CE !cA_a !connect or with CA cA_b through the event DE_to_CE !cA_b !disconnect. In all cases, the DA shrieks both the identifier of the CA and the message it wishes to send. For any action to be executed, one CA needs to agree on the first experiment (the CA identifier) and accept the second one (the message).

3.3.2 Analysis & Decisions

During the analysis phase, the LOTOS specifier acquires a global picture of the system. This knowledge is used to take decisions on the specificity of the mapping. Decisions are taken regarding the translation of UCM into LOTOS. They relate to the representation of static and dynamic stubs, plug-ins, data structures, databases, and agents (DA, CA, etc). The system contains the following types of components:

- **Agents** – are DAs, CAs, LAs and Call Objects instantiations.
- **Stubs** – are all the possible static and dynamic stubs used in the agents.
- **Plug-ins** – are all possible plug-ins that can be plugged in the dynamic stubs.
- **Database** – refers to all databases grouped in a single one.

Components are modeled as processes. The database and agents are instantiations at the behavior level of the specification. Stub and plug-in instantiations are at lower levels, inside agent processes. In essence, the architecture of the LOTOS specification reflects that of the UCM model. We have defined guidelines, presented below, to construct LOTOS specifications from UCM.

General Translation Guidelines

- Start points and end points are usually represented by LOTOS gates in the prototype. They can then be controlled and observed during the validation.

- UCM responsibilities are also represented as gates, sometimes with additional message exchanges (to insure causality across components).

- LOTOS gates representing UCM responsibilities and channels that are not observable are hidden through the hide operator.

- UCM components are represented as processes synchronized on their shared channels/gates. The structure is specified mostly in a resource-oriented style [39], with multi-way synchronization (|[...]|) and interleaving (|||) operators.

- Containment of components is maintained. If a component, represented by a LOTOS process, contains sub-components, then the processes representing these are instantiated (and possibly defined) within the former process.

- If multiple path segments (possibly from different UCM scenarios) cross one component, they are integrated together in the same LOTOS process, often as alternatives.

- Elementary processes are specified mostly in a state-oriented style, with choice ([]) and action prefix (;) operators, and with guarded behaviors ([...] $\rightarrow$).

- Abstract data types are used to represent operations, and conditions (LOTOS guard expressions).

- Symmetry is enforced in synchronized actions: actions in one component/process must be mirrored in the other synchronized processes, unless locally hidden.

- Components with stubs have sub-processes, one for each stub. For dynamic stubs, these processes specify the selection policy, i.e. the type of composition between the possible plug-ins.

- Dynamic stubs may have multiple sub-processes, one for each plug-in, whereas static stubs are refined directly by the process representing their only plug-in.

- Stub processes receive a list of entry/exit points as input and then output another such list upon termination.

- Time is not intrinsic to LOTOS. Timers are represented as processes and timeouts are represented using actions. A timeout is simulated by the execution of its corresponding action.

- OR-Forks are represented using the *choice* operator.

- OR-Join, AND-Fork/Join and rendezvous are represented using the LOTOS synchronization mechanisms.

Stubs & Plug-ins

Stubs and plug-ins are elements of agents. DA, CA, LA and CallObject are agents composed of several static and dynamic stubs. These stubs can in turn contain other stubs and so on. Stubs and plug-ins are translated into LOTOS processes and instantiations. A static stub is represented as a process. A dynamic stub is represented by the instantiation of a process of the same name. This process handles the instantiation of one or many of the sub-processes based on selection policies; each sub-process corresponding to a plug-in of the dynamic stub. Those choices result in a LOTOS specification which is close to the UCM model. It also makes the translation of features a straightforward technique, consisting in following the UCM model and choosing adequate plug-ins.

Agents & Database

Agents and Database are at the same level, namely the root of the system. The database is represented as a process that is initialized at the time of its instantiation. All agents are uniquely identified.

DAs are static: for any particular DA, there exists one instance. Each DA process is instantiated with values corresponding to its identifier (ID) and the user with whom it is associated. CAs, LAs and Call Objects are managed in a more dynamic fashion. CAs and LAs can be involved in several communication scenarios at a time. To permit this, CA and LA processes are designed such that there may be many instances of them at a give time. In addition to their unique *process* identifier, each *instance* of a given CA or LA process is also uniquely identified. These agents are therefore identified by a tuple {*id*, *instance*}.

The first parameter, *id*, is the main identifier, much like DA's ID; the second parameter, *instance*, represents a specific instance of the CA or the LA process. Immediately after the initialization of the system, there exists only one instance of each agent. When a CA is contacted, it handles the communication and concurrently replicates itself into a new instance. This new instance satisfies future requests. The synchronization between distinct instances is done using *gate splitting*. Hence, two CAs can use the same LA without conflict; that is, each CA communicates with a different LOTOS instance of the same LA.

Call Objects are managed similarly. A Call Object process is identified solely by its instance identifier. Immediately after the initialization of the system, there is no Call Object process in existence. The creation of Call Object instances is handled dynamically by a special process (Call Object Creator) which initially waits for requests. When a LA asks for the creation of a call, the Call Object Creator creates a new Call Object instance and gives it a specific instance identifier to avoid the conflicts between existing instances.

3.3.3 Features Integration

Once the basic LOTOS specification is completed and conforms to the UCM model, features can be integrated. Ten features have been implemented: *Call Forward Always (CFA)*, *Call Forward on Busy (CFB)*, *Outgoing Call Screening (OCS)*, *Call Hold (CH)*, *Call Pickup (CP)*, *Call Transfer (CT)*, *Auto Recall (AR)*, *Time Reminder (TMR)* and *Call Waiting (CW)*. The LOTOS specification being close to the Use Case Maps model, implementing a feature consists in using its plug-ins and modifying the dynamic stubs composing the feature.

The system contains about one hundred and ten stubs and locating them is a tedious task. The difficulty increases with the depth of recursion. The use of specialized techniques reduces this difficulty. One such technique consists in representing the components as a graph, using tools like *GraphViz (Graph Visualization*, a tool from Bell Labs). For any feature, the UCM are browsed while taking note of the components used and their relationships. This textual description is fed by *GraphViz*, which generates a graph from it. The graphical representation of the components helps to identify the parts that need be modified.

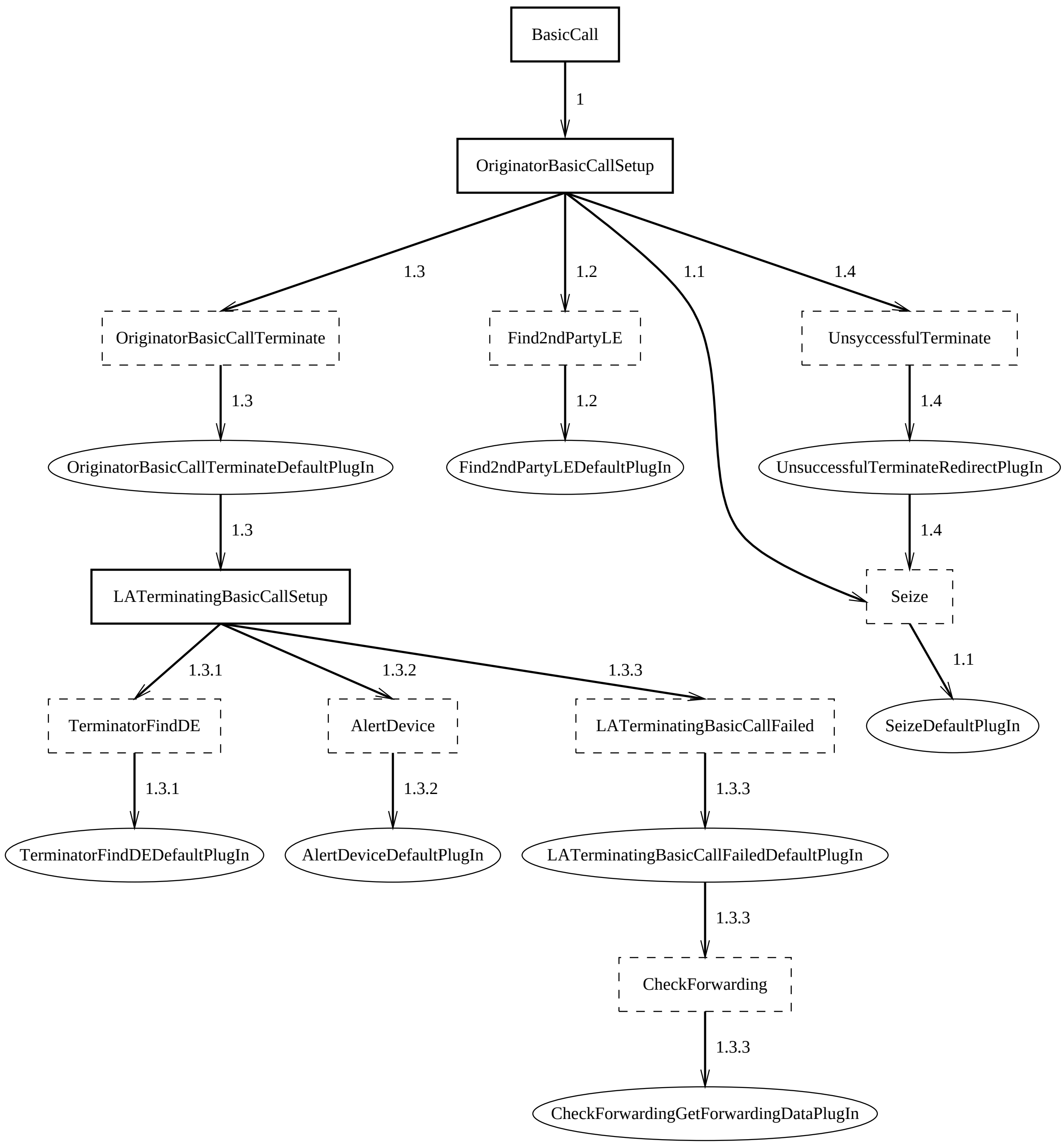

Figure 3.11: Graph of Stubs & Plug-Ins of CFB

Fig. 3.11 illustrates the graph generated by *GraphViz* for *Call Forward on Busy*, where static stubs are represented by solid boxes, dynamic stubs by dashed boxes and plug-ins by ellipses.

The LOTOS specification is 2410 lines long. This breaks down into 450 lines of Abstract Data Types (ADT) and 1960 lines of behavior.

3.4 In Summary

In this chapter, we presented the specification part of our project. We presented the Use Case Maps notation as well as the basic LOTOS concepts. In addition, we presented the system and the features provided by Mitel Corp. as well as guidelines for the translation of the UCM model into a LOTOS specification. This permits expressing requirements in a formal and executable model and, thus, to perform tool-aided validation of the model.

Chapter 4

Identification of Incoherences – Theory

As mentioned in chapter 2, identifying feature interactions is not a trivial task and it is believed that formal and automatic methods are helpful. Moreover, identifying interactions early in the development process reduces the cost of testing. We propose a filtering method that is based on the principle of logic and resolution proof [40]. Our method allows to identify incoherences corresponding to potential pair-wise feature interactions at the requirements stage.

This method deals with features without considering the system as a whole specification. It only identifies incoherences on which the tester should focus. We define a simple formal notation for the description of features using information obtained from requirements. We define and verify formulas relative to specific types of incoherences that correspond to possible pair-wise feature interactions.

4.1 Features Representation

A feature implements a functionality that applies to specific states or situations of a call process. When a call process is in a state for which a given feature is defined, the functionality of the feature is used instead of, or in addition to, the functionality provided by the base system. Since a call serves the communication purposes of users, features are associated, or bound, to users. We see the evolution of a call process as a sequence of events bringing the system from a state to another. Thus, the activation of a feature, and the use of its functionality, depends on the state in which the system is and the event that occurs. A feature in a certain state is triggered by some specific event and the use of such a feature leads to some specific results according to the functionality provided.

4.1.1 Basic Principles

A feature implementing a functionality that presents behaviors at different states is broken into several description parts, each one representing a different behavior. Hence, the name of a description part is composed of the name of the feature itself and the part concerned. Each description part is decomposed into 4 specific groups of properties: *pre-conditions*, *triggering* *events*, *results* and *constraints*. Properties are composed of a name and a set of elements that can be variables or sub-properties.

Pre-conditions – represent the mandatory conditions for the activation of the feature. These conditions describe the state in which the system must be before the activation of the feature can be considered.

Triggering events – represent the action(s) triggering the feature

Results – are the actions produced by the execution of the feature and the state in which the system is after such execution.

Constraints – are the restrictions relative to the variables used in pre-conditions, triggering events and results describing the feature.

4.1.2 Formalism

As mentioned, properties are composed of a name and a set of elements. The set of elements can be empty (i.e. the property is only a name) or can contain a list of variables representing the users concerned with the property. In order to distinguish names from variables, property names start with a lowercase while variable names start with an uppercase. The grammar defining the syntax of a property is the following:

```
PROPERTY   ::=   NAME "(" VARLIST ")" | NAME
VARLIST    ::=   VAR | VARLIST "," VAR
NAME       ::=   [a-z][A-Za-z0-9]*
VAR        ::=   [A-Z][A-Za-z0-9]*
```

A property is formally denoted $\gamma(v_1, ..., v_i)$, $i > 0$, where γ stands for the name of the property and the list $v_1, ..., v_i$ are variables representing entities concerned with the property. Property names are not limited by any specific rule. They only need to be homogeneous with respect to different features. Two names are however reserved for specific mandatory properties; `subs` and `concerns`, defined below:

- `subs(U, F)` – represents a user, denoted by the variable U, that subscribes to a feature, denoted by variable F. For instance, the fact that the user Bob subscribes to the feature Call Waiting, represented by the acronym `cw`, is denoted: `subs(bob, cw)`, where $U =$ bob and $F =$ cw.

- `concerns(U, F)` – stipulates that the usual behavior of user U can be influenced by the 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 that case, the user concerned is not the subscriber but the one on hold. Alice being concerned with Call Waiting is denoted `concerns(alice, cw)`.

A feature X is denoted $\mathcal{F}_X$ and can be broken into several description parts, each one relative to one of the behaviors of the functionality. 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 implementing 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 $\mathcal{F}_X = \{\mathcal{D}_{X,1}, \mathcal{D}_{X,2}, \mathcal{D}_{X,3}\}$.

As for a description part, each of the four groups of properties composing it is denoted by a letter that corresponds to 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 groups are denoted as follows:

- The set of pre-conditions is denoted $\mathcal{P}_{X,m}$. It is an ordered set of properties formally defined as: $\mathcal{P}_{X,m} = \{\gamma_1(\upsilon_{1,1}, ..., \upsilon_{1,i}), ..., \gamma_j(\upsilon_{j,1}, ..., \upsilon_{j,i})\}$, $i, j > 0$, where $\gamma_1, ..., \gamma_j$ are properties representing pre-conditions. Pre-conditions must always contain the two properties `subs` and `concerns`, denoting the subscriber and the user influenced by the functionality (in most cases, the subscriber itself).

- Following the same pattern, the set of triggering events is an ordered set denoted $\mathcal{T}_{X,m}$, and we express $\mathcal{T}_{X,m} = \{\gamma_1(\upsilon_{1,1}, ..., \upsilon_{1,i}), ..., \gamma_j(\upsilon_{j,1}, ..., \upsilon_{j,i})\}$, $i, j > 0$, where $\gamma_1, ..., \gamma_j$ are properties representing triggering events.

- Based again on the same pattern, the set of results is an ordered set denoted $\mathcal{R}_{X,m}$, and we express $\mathcal{R}_{X,m} = \{\gamma_1(\upsilon_{1,1}, ..., \upsilon_{1,i}), ..., \gamma_j(\upsilon_{j,1}, ..., \upsilon_{j,i})\}$, $i, j > 0$, where $\gamma_1, ..., \gamma_j$ are properties representing results.

- The set of constraints does not contain properties but restrictions. It identifies restrictions on the variables involved, which represent users. This set is denoted $\mathcal{C}_{X,m}$, and, $\mathcal{C}_{X,m} = \{(\upsilon_1 \Re_1 \upsilon_2), ..., (\upsilon_i \Re_j \upsilon_{i+1})\}$, $i, j \geq 0$, where $\upsilon_1, ..., \upsilon_{i+1}$ are the variables used in $\mathcal{P}_{X,m}$, $\mathcal{T}_{X,m}$ and $\mathcal{R}_{X,m}$, and, $\Re_1, ..., \Re_j$ represent the relations between variables regarding equality ($=$) or inequality ($\neq$). Such constraints can represent the fact that a caller must be different from the callee, or that they must be the same user.

4.1.3 Example

Let us consider an example of feature definition: `Call Waiting`. `Call Waiting` implements the following functionality: when the subscriber is busy and receives a call, the caller is put on hold and the subscriber receives a notification signal. Then, pressing a special button, the subscriber can switch from the talking party to the held party and vice-versa. If a call request is received by the held party, a busy notification is sent to the caller. We distinguish three behaviors and thus, three description parts that are:

1. The subscriber is talking, receives a call and gets a notification.

2. The subscriber is holding a caller and switches from the talking party to the held one.

3. The held party receives a call request and sends a busy indication.

In order to formalize the description of this feature, we use the acronym `cw` to represent Call Waiting and we establish the list of properties corresponding to the requirements above:

- `subs(A, cw)` – User A subscribes to Call Waiting.
- `concerns(B, cw)` – Call Waiting concerns user B.
- `busy(B)` – User B is busy.
- `talk(A, B)` – User A is talking with user B.
- `call(A, C)` – Call attempt from a user A to another user C.
- `hold(A, C)` – User A holds user C.
- `cw_notify(A)` – User A receives a *hold* notification.
- `flash(A)` – User A presses flash button to switch between parties.
- `busy_ind(C, D)` – User C sends a busy indication to user D.

We formally express the feature description parts 1, 2 and 3 as:

- $\mathcal{D}_{cw,1}$, Call Waiting, description part 1: subscriber A receives a call from C.

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

- $\mathcal{D}_{cw,2}$, Call Waiting, description part 2: subscriber A switches between parties.

$$\begin{aligned}
\mathcal{P}_{cw,2} &= \{ \text{ subs(A, cw), concerns(A, cw), busy(A), talk(A, B), hold(A, C) } \} \\
\mathcal{T}_{cw,2} &= \{ \text{ flash(A) } \} \\
\mathcal{R}_{cw,2} &= \{ \text{ hold(A, B), talk(A, C) } \} \\
\mathcal{C}_{cw,2} &= \{ \text{ (A} \neq \text{B), (A} \neq \text{C), (B} \neq \text{C) } \}
\end{aligned}$$

- $\mathcal{D}_{cw,3}$, Call Waiting, description part 3: held party C receives a call from D.

$$\begin{aligned}
\mathcal{P}_{cw,3} &= \{ \text{ subs(A, cw), concerns(C, cw), busy(A), talk(A, B), hold(A, C) } \} \\
\mathcal{T}_{cw,3} &= \{ \text{ call(D, C) } \} \\
\mathcal{R}_{cw,3} &= \{ \text{ busy_ind(C, D) } \} \\
\mathcal{C}_{cw,3} &= \{ \text{ (A} \neq \text{B), (A} \neq \text{C), (A} \neq \text{D), (B} \neq \text{C), (B} \neq \text{D), (C} \neq \text{D) } \}
\end{aligned}$$

Parts 1 & 2 of the feature concern the subscriber since the functionality is added to him. In part 3, the user concerned is not the subscriber but the held party. We consider that the held party is the concerned user because his behavior is modified by the feature (which is external to him) in case of an incoming call. All users are assumed to be different because we are addressing only the general case. Particular cases such as the one where a user calls itself were not considered. Such cases can be explicitly considered by using appropriate constraints.

4.2 Incoherences

The identification of incoherences between two features is based on the identification of specific incoherences between their set of properties. We represent these incoherences in the form of rules and check whether one or more of the rules can be satisfied or not by binding variables of feature descriptions to users. If a rule is satisfied, an incoherence is identified.

Two properties can present a contradiction. When formally defining a feature, contradictory pairs of properties must be identified and listed for future analysis purposes. A contradiction taking place between two properties is stated using a binary relation denoted $\mathcal{K}$ $(\gamma_1(\upsilon_{1,1}, ..., \upsilon_{1,i}), \gamma_2(\upsilon_{2,1}, ..., \upsilon_{2,j}))$, i, j > 1, where γ_1 and γ_2 are the two properties considered and υ_1, υ_2 are the variables used to represent users.

Variables allow to express particular restrictions such as the users concerned with two properties must be the same. For instance, let us say that `busy(X)` denotes that a user, represented by the variable X, is busy. And let us say that `idle(Y)` denotes that a user, represented by the variable Y, is idle. A user being idle and busy at the same time not being permitted can be expressed as contradiction: $\mathcal{K}$(`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`.

Unbound properties are properties for which variables have no fixed values. As mentioned, they are represented with their name and variables such as: $\gamma_1(\upsilon_1, ..., \upsilon_i)$. To simplify the notation, properties for which variables are bound (have been given a value) are represented without their variables. An unbound property, denoted $\gamma_1(\upsilon_1, ..., \upsilon_i)$ is denoted π_1 when bound.

Note that, in order to simplify the notation, we use the same names $\mathcal{F}_X$, $\mathcal{D}_{X,m}$, etc. to denote sets of properties over variables or over constants.

4.2.1 Definitions

In order to formalize incoherences identification, a few definitions need to be stated:

- We define the combination of two feature descriptions as the ordered pair of the two descriptions where the variables of a description are kept 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 $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$.

- The same user cannot subscribe to the same feature more than once. We express this fact as a general contradiction denoted $\mathcal{K}$(subs(A, X), subs(A, X)) where the variable A represents the user and variable X represents the name of the feature.

- We denote $\mathcal{S}(\mathcal{C}_{X,m})$ the fact that all constraints of the constraint set $\mathcal{C}_{X,m}$ are satisfied.

- We define Υ the finite set of all possible users that can be used to bind variables in properties as $\Upsilon = \{\upsilon_1, ..., \upsilon_n\}$, for some n representing the number of users.

- Two distinct ordered sets $\mathcal{E}_1$ and $\mathcal{E}_2$ present a contradiction if there exists $\pi_1 \in \mathcal{E}_1$ and a $\pi_2 \in \mathcal{E}_2$ such that $\mathcal{K}(\pi_1, \pi_2)$ is satisfied. Note that this is decidable because only a finite number of variables and values are in discussion.

- Two distinct ordered sets $\mathcal{E}_1$ and $\mathcal{E}_2$ are equal to each other ($\mathcal{E}_1 = \mathcal{E}_2$) if they contain the same elements in the same order. Otherwise, they are different ($\mathcal{E}_1 \neq \mathcal{E}_2$).

Incoherences refer to incoherences between behaviors. Hence, an incoherence between two features $\mathcal{F}_X$ and $\mathcal{F}_Y$ is characterized by an incoherence present between a description part $\mathcal{D}_{X,m}$ of $\mathcal{F}_X$ and another description part $\mathcal{D}_{Y,n}$ of $\mathcal{F}_Y$. Different types of incoherences can be identified. An incoherence can be characterized by non determinism, contradiction, or loops. Considering pair-wise feature combinations, we identify six global types of incoherences. The types are gathered in two main categories:

- Direct incoherences – for which identification rules are presented in sub-section 4.2.2
- Transitive incoherences – for which identification rules are presented in sub-section 4.2.3

Variables are bound to users. The number of users to use depends on the number of variables used in the properties of feature descriptions. It must be possible to bind all variables contained in the properties of a feature description to different values. In order to fulfill this requirement, the number of required users may reach up to twice the number of variables of the description that contains the greater number of variables. However, in most cases, only half of these users may be sufficient.

4.2.2 Direct Incoherence Rules

Direct incoherences are present when two description parts $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ are linked together, present the same triggers but lead to different or contradictory results. In both cases, non determinism arises since two possible results, valid from each feature's point of view, are possible. Two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ can be linked together and present an incoherence for the following reasons:

1. The same user subscribes to both features $\mathcal{F}_X$ and $\mathcal{F}_Y$. An incoherence can be present if:

   1a. Features can be triggered by the same event and present results that are different but not contradictory.

   1b. Features can be triggered by the same event and present contradictory results.

2. Different users U and V respectively subscribe to $\mathcal{F}_X$ and $\mathcal{F}_Y$ and U is concerned with both features $\mathcal{F}_X$ and $\mathcal{F}_Y$. Again, an incoherence can be present for two reasons:

   2a. Features can be triggered by the same event and present results that are different but not contradictory.

   2b. Features can be triggered by the same event and present contradictory results.

All cases (1a, 1b, 2a, 2b) result in non-determinism except that, in cases 1a and 2a, results are different and do not present contradictions while in cases 1b and 2b, results are different and present a contradiction. We distinguish four rules for direct incoherences, each one associated to a case: direct incoherence rule #d1, associated to case 1a; direct incoherence rule #d2, associated to case 1b; direct incoherence rule #d3, associated to case 2a; direct incoherence rule #d4, associated to case 2b.

Direct Incoherence Rule #d1

Rule #d1 identifies incoherences between features subscribed by the same user, triggered by the same event and yielding different but not contradictory results. Let us consider two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ combined in a pair $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$. An incoherence is present if, from Υ, the set of all possible users, it is possible to find a binding of property variables such that the following holds:

1. The same user subscribes to both $\mathcal{F}_X$ and $\mathcal{F}_Y$,
2. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ do not present a contradiction,
3. $\mathcal{T}_{X,m}$ and $\mathcal{T}_{Y,n}$ are the same,
4. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ are different,
5. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ do not present a contradiction,
6. $\mathcal{S}(\mathcal{C}_{X,m})$ and $\mathcal{S}(\mathcal{C}_{Y,n})$, constraints of both descriptions are satisfied.

Due to its configuration (features are subscribed by the same user, have the same triggering events and different results), the incoherences identified by this rule are *symmetric*. If the rule identifies an incoherence in $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$, it also identifies the same incoherence in $|\mathcal{D}_{Y,n} \bullet \mathcal{D}_{X,m}|$. Similarly, if no incoherence is found in one pair, none will be found in the other one. Hence, the rule only needs to be applied to one of the two pairs. The rule is formally represented as follows:

Variables of properties are bound using values provided by Υ such that:

1. $\exists\ \upsilon \in \Upsilon \mid (\text{subs}(\upsilon,\ X) \in \mathcal{P}_{X,m} \wedge \text{subs}(\upsilon,\ Y) \in \mathcal{P}_{Y,n})$
2. $\nexists(\pi_1 \in \mathcal{P}_{X,m},\ \pi_2 \in \mathcal{P}_{Y,n}) \mid \mathcal{K}(\pi_1,\ \pi_2)$
3. $\mathcal{T}_{X,m} = \mathcal{T}_{Y,n}$
4. $\mathcal{R}_{X,m} \neq \mathcal{R}_{Y,n}$
5. $\nexists(\pi_3 \in \mathcal{R}_{X,m},\ \pi_4 \in \mathcal{R}_{Y,n}) \mid \mathcal{K}(\pi_3,\ \pi_4)$
6. $\mathcal{S}(\mathcal{C}_{X,m}) \wedge \mathcal{S}(\mathcal{C}_{Y,n})$

As an example, let us consider two features: `Call Forward on Busy` and `Incoming Call Screening`. `Call Forward on Busy` implements the possibility, for the subscriber, to forward incoming calls to another party when busy. `Incoming Call Screening` allows the subscriber to establish a screening list and to deny calls incoming from users contained in this list. The list of properties corresponding to the requirements follows:

- `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 feature only implements one functionality. Hence, one description part is sufficient to formalize each feature. In order to consider the general cases of the use of `Call Forward on Busy` and `Incoming Call Screening`, all variables involved in a feature description must have different values. This eliminates particular cases such as those where a user forwards all its calls to himself when he is busy. We use the acronyms `cfb` to denote `Call Forward on Busy` and `ics` to denote `Incoming Call Screening`; the number 1 denotes the (single) description part of each feature:

- $\mathcal{D}_{cfb,1}$ Call Forward on Busy, description part 1, subscriber B is busy and receives a call from A:
  $\mathcal{P}_{cfb,1}$ = { subs(B, cfb), concerns(B, cfb), cfb(C), busy(B) }
  $\mathcal{T}_{cfb,1}$ = { call(A, B) }
  $\mathcal{R}_{cfb,1}$ = { redirected(A, C), call(A, C) }
  $\mathcal{C}_{cfb,1}$ = { (A $\neq$ B), (A $\neq$ C), (B $\neq$ C) }
- $\mathcal{D}_{ics,1}$ Incoming Call Screening, description part 1, screened party C calls B:
  $\mathcal{P}_{ics,1}$ = { subs(B, ics), concerns(B, ics), ics_list(A) }
  $\mathcal{T}_{ics,1}$ = { call(A, B) }
  $\mathcal{R}_{ics,1}$ = { deny_call(B, A), call_denied(B, A) }
  $\mathcal{C}_{ics,1}$ = { (A $\neq$ B) }

Performing pair-wise combination over $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$, we 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. Let us use the notation $X \leftarrow val$ to denote the binding of variable X to value val and let us consider three users: *Alice*, *Bob*, *Carol*. We bind the variables of the properties of $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ to the following values:

- $\mathcal{D}_{cfb,1}$ – $A \leftarrow$ alice, $B \leftarrow$ bob, $C \leftarrow$ carol.
  $\mathcal{P}_{cfb,1}$ = { subs(bob, cfb), concerns(bob, cfb), cfb(carol), busy(bob) }
  $\mathcal{T}_{cfb,1}$ = { call(alice, bob) }
  $\mathcal{R}_{cfb,1}$ = { redirected(alice, carol), call(alice, carol) }
  $\mathcal{C}_{cfb,1}$ = { (alice $\neq$ bob), (alice $\neq$ carol), (bob $\neq$ carol) }
- $\mathcal{D}_{ics,1}$ – $A \leftarrow$ alice, $B \leftarrow$ bob.
  $\mathcal{P}_{ics,1}$ = { subs(bob, ics), concerns(bob, ics), ics_list(alice) }
  $\mathcal{T}_{ics,1}$ = { call(alice, bob) }
  $\mathcal{R}_{ics,1}$ = { deny_call(bob, alice), call_denied(bob, alice) }
  $\mathcal{C}_{ics,1}$ = { (alice $\neq$ bob) }

Rule #d1 is satisfied for this binding since:

1. subs(bob, cfb) $\in \mathcal{P}_{cfb,1} \wedge$ subs(bob, ics) $\in \mathcal{P}_{ics,1}$
2. $\nexists\, \pi_1 \in \mathcal{P}_{cfb,1},\, \pi_2 \in \mathcal{P}_{ics,1} \mid \mathcal{K}(\pi_1, \pi_2)$
3. $\mathcal{T}_{cfb,1} = \mathcal{T}_{ics,1}$
4. $\mathcal{R}_{cfb,1} \neq \mathcal{R}_{ics,1}$
5. $\nexists\, \pi_3 \in \mathcal{R}_{cfb,1},\, \pi_4 \in \mathcal{R}_{ics,1} \mid \mathcal{K}(\pi_3, \pi_4)$
6. $\mathcal{S}(\mathcal{C}_{cfb,1}) \wedge \mathcal{S}(\mathcal{C}_{ics,1})$

If Alice calls Bob (call(alice, bob)), `Call Forward on Busy` should be activated and Alice should be redirected to Carol (redirected(alice, carol)) and call Carol (call(alice, carol)). At the same time, `Incoming Call Screening` should also be activated and Alice's call should be denied (deny_call(bob, alice)). We conclude that $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ present an incoherence characterized by non-determinism.

Direct Incoherence Rule #d2

Direct incoherence rule #d2 identifies incoherences characterized by contradictory results. Similarly to rule #d1, it identifies *symmetric* incoherences. An incoherence is present if, from Υ, it is possible to find a binding of property variables such that the following holds:

1. The same user subscribes to both $\mathcal{F}_X$ and $\mathcal{F}_Y$,
2. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ do not present a contradiction,
3. $\mathcal{T}_{X,m}$ and $\mathcal{T}_{Y,n}$ are the same,
4. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ present a contradiction,
5. $\mathcal{S}(\mathcal{C}_{X,m})$ and $\mathcal{S}(\mathcal{C}_{Y,n})$, constraints of both descriptions are satisfied.

The rule is formally defined as follows:

Variables of properties are bound using values provided by Υ such that:

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

As an example, let us consider the feature definition of `Call Waiting` given in section 4.1.3 and combine it with the feature definition of `Incoming Call Screening` given in section 4.2.2. In addition to the list of properties previously defined for these two features, we consider that a user cannot hold a call and deny it at the same time. This contradiction is expressed by the following relation:

- $\mathcal{K}$(hold(B, A), deny_call(B, A)) – A user B cannot, at the same time, hold and deny a call incoming from user A.

Let us consider that the system contains 3 possible users: *Alice*, *Bob*, *Carol* and let us bind variables of $\mathcal{D}_{cw,1}$ and $\mathcal{D}_{ics,1}$ to users such that their constraints are satisfied. A binding of properties variables of both descriptions satisfying the rule #d2 is:

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

  $\mathcal{P}_{cw,1}$ = { subs(bob, cw), concerns(bob, cw), busy(bob) talk(bob, carol) }
  $\mathcal{T}_{cw,1}$ = { call(alice, bob) }
  $\mathcal{R}_{cw,1}$ = { hold(bob, alice), cw_notify(bob) }
  $\mathcal{C}_{cw,1}$ = { (alice $\neq$ bob), (alice $\neq$ carol), (bob $\neq$ carol) }

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

  $\mathcal{P}_{ics,1}$ = { subs(bob, ics), concerns(bob, ics), ics_list(alice) }
  $\mathcal{T}_{ics,1}$ = { call(alice, bob) }
  $\mathcal{R}_{ics,1}$ = { deny_call(bob, alice), call_denied(bob, alice) }
  $\mathcal{C}_{ics,1}$ = { (alice $\neq$ bob) }

The rule is satisfied since:

1. subs(bob, cw) $\in \mathcal{P}_{cw,1} \wedge$ subs(bob, ics) $\in \mathcal{P}_{ics,1}$
2. $\nexists\, \pi_1 \in \mathcal{P}_{cw,1},\, \pi_2 \in \mathcal{P}_{ics,1} \mid \mathcal{K}(\pi_1, \pi_2)$
3. $\mathcal{T}_{cw,1} = \mathcal{T}_{ics,1}$
4. $\exists\, \pi_3 \in \mathcal{R}_{cw,1},\, \pi_4 \in \mathcal{R}_{ics,1} \mid \mathcal{K}(\pi_3, \pi_4)$
5. $\mathcal{S}(\mathcal{C}_{cw,1}) \wedge \mathcal{S}(\mathcal{C}_{ics,1})$

If Alice calls Bob (call(alice, bob)), `Call Waiting` should be activated, and hold Alice (hold(bob, alice)). At the same time, `Incoming Call Screening` should also be activated and deny the call incoming from Alice (deny_call(bob, alice)) since Alice is in the screening list. If Alice is held by `Call Waiting`, she will be able to talk to Bob later, which should not happen. Hence, $\mathcal{D}_{cw,1}$ and $\mathcal{D}_{ics,1}$ present an incoherence characterized by a contradiction.

Direct Incoherence Rule #d3

Rule #d3 identifies incoherences taking place between features subscribed by different users and both concerning the subscriber of one of the features. An incoherence is present if the following characteristics hold:

1. A user υ_1 subscribes to $\mathcal{F}_X$,
   a different user υ_2 subscribes to $\mathcal{F}_Y$,
   the user υ_2 is concerned with both descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$,

2. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ do not present a contradiction,

3. $\mathcal{T}_{X,m}$ and $\mathcal{T}_{Y,n}$ are the same,

4. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ are different,

5. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ do not present a contradiction,

6. $\mathcal{S}(\mathcal{C}_{X,m})$ and $\mathcal{S}(\mathcal{C}_{Y,n})$, constraints of both features are satisfied.

Rule #d3 identifies similar incoherences as rule #d1 where $\mathcal{T}_{X,m} = \mathcal{T}_{Y,n}$ and $\mathcal{R}_{X,m} \neq \mathcal{R}_{Y,n}$. Given two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$, the rule stipulates that $\mathcal{D}_{X,m}$ must have a user υ_1 as a subscriber and must concern a different user υ_2 and that $\mathcal{D}_{Y,n}$ must have υ_2 as a subscriber and concerned user.

Identifying an incoherence in $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$ does not insure that the same incoherence is also present in $|\mathcal{D}_{Y,n} \bullet \mathcal{D}_{X,m}|$, which means that, in order to consider all cases, incoherence rule #d3 must be applied to both combinations. In other words, contrary to rule #d1, rule #d3 is not symmetric.

The rule is formally defined as follows:

Variables of properties are bound using values provided by Υ such that:

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

2. $\nexists(\pi_1 \in \mathcal{P}_{X,m},\, \pi_2 \in \mathcal{P}_{Y,n}) \mid \mathcal{K}(\pi_1,\, \pi_2)$

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

4. $\mathcal{R}_{X,m} \neq \mathcal{R}_{Y,n}$

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

6. $\mathcal{S}(\mathcal{C}_{X,m}) \wedge \mathcal{S}(\mathcal{C}_{Y,n})$

As an example, let us consider `Call Waiting` (defined in section 4.1.3) and `Call Forward on Busy` (defined in section 4.2.2). We observe that $|\mathcal{D}_{cw,3} \bullet \mathcal{D}_{cfb,1}|$ presents an incoherence since it is possible to find a binding of variables that satisfies incoherence rule #d3. Let us consider 4 possible users: *Alice*, *Bob*, *Carol*, *Dave*, and use them to bind variables of $\mathcal{D}_{cw,3}$ and $\mathcal{D}_{cfb,1}$ such that their constraints are satisfied and such that rule #d3 is also satisfied:

- $\mathcal{D}_{cw,3}$ – $A \leftarrow$ alice, $B \leftarrow$ bob, $C \leftarrow$ carol, $D \leftarrow$ dave.

$$\begin{aligned}
\mathcal{P}_{cw,3} &= \{ \text{ subs(alice, cw), concerns(carol, cw), busy(alice),} \\
&\quad\ \text{talk(alice, bob), hold(alice, carol) } \} \\
\mathcal{T}_{cw,3} &= \{ \text{ call(dave, carol) } \} \\
\mathcal{R}_{cw,3} &= \{ \text{ busy_ind(carol, dave) } \} \\
\mathcal{C}_{cw,3} &= \{ \text{ (alice} \neq \text{bob), (alice} \neq \text{carol), (alice} \neq \text{dave), (bob} \neq \text{carol),} \\
&\quad\ \text{(bob} \neq \text{dave), (carol} \neq \text{dave) } \}
\end{aligned}$$

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

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

The rule is satisfied since:

1. (subs(alice, cw) $\in \mathcal{P}_{cw,3} \wedge$ concerns(carol, cw) $\in \mathcal{P}_{cfb,1}$) $\wedge$
   (subs(carol, cfb) $\in \mathcal{P}_{cw,3} \wedge$ concerns(carol, cfb) $\in \mathcal{P}_{cfb,1}$)
2. $\nexists\, \pi_1 \in \mathcal{P}_{cw,3},\, \pi_2 \in \mathcal{P}_{cfb,1} \mid \mathcal{K}(\pi_1,\, \pi_2)$
3. $\mathcal{T}_{cw,3} = \mathcal{T}_{cfb,1}$
4. $\mathcal{R}_{cw,3} \neq \mathcal{T}_{cfb,1}$
5. $\nexists\, \pi_3 \in \mathcal{R}_{cw,3},\, \pi_4 \in \mathcal{R}_{cfb,1} \mid \mathcal{K}(\pi_3,\, \pi_4)$
6. $\mathcal{S}(\mathcal{C}_{cw,3}) \wedge \mathcal{S}(\mathcal{C}_{cfb,1})$

Features have different subscribers but concern the same user. Carol is busy and held by Alice. When Carol receives a call (call(dave, carol)), `Call Waiting`, on Alice side, informs that the switch should send a busy indication to the caller (busy_ind(carol, dave)), while `Call Forward on Busy`, on her side, informs that the call should be forwarded (redirected(dave, alice)). We face a non determinism since both feature parts are triggered by the same event and lead to different results. Thus, $\mathcal{D}_{cw,3}$ and $\mathcal{D}_{cfb,1}$ present an incoherence characterized by a non-determinism.

Direct Incoherence Rule #d4

Similarly to direct incoherence rule #d3, rule #d4 identifies incoherences present between features subscribed by different users and concerning the same user. Again, for partitioning issues, this rule is similar to #d2 since it concerns the set of incoherences characterized by contradictory results. Also, as for #d3, due to its configuration, this rule does not identify *symmetric* incoherences.

As for all other rules defined before, we consider two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$. An incoherence is present if, from Υ, the set of all possible users, it is possible to find a binding of property variables such that the combination of the two features considered are the following:

1. A user υ_1 subscribes to $\mathcal{F}_X$,
   a different user υ_2 subscribes to $\mathcal{F}_Y$,
   the user υ_2 is concerned with both descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$,
2. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ do not present a contradiction,
3. $\mathcal{T}_{X,m}$ and $\mathcal{T}_{Y,n}$ are the same,
4. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ present a contradiction,
5. $\mathcal{S}(\mathcal{C}_{X,m})$ and $\mathcal{S}(\mathcal{C}_{Y,n})$, constraints of both features are satisfied.

The rule is formally defined as follows:

Variables of properties are bound using values provided by Υ such that:

1. $\exists\ \upsilon_1,\ \upsilon_2 \in \Upsilon,\ \upsilon_1 \neq \upsilon_2 \mid$
   $(\text{subs}(\upsilon_1,\ X) \in \mathcal{P}_{X,m} \wedge \text{concerns}(\upsilon_2,\ X) \in \mathcal{P}_{X,m}) \wedge$
   $(\text{subs}(\upsilon_2,\ X) \in \mathcal{P}_{Y,n} \wedge \text{concerns}(\upsilon_2,\ Y) \in \mathcal{P}_{Y,n})$
2. $\nexists(\pi_1 \in \mathcal{P}_{X,m},\ \pi_2 \in \mathcal{P}_{Y,n}) \mid \mathcal{K}(\pi_1,\ \pi_2)$
3. $\mathcal{T}_{X,m} = \mathcal{T}_{Y,n}$
4. $\exists(\pi_3 \in \mathcal{R}_{X,m},\ \pi_4 \in \mathcal{R}_{Y,n}) \mid \mathcal{K}(\pi_3,\ \pi_4)$
5. $\mathcal{S}(\mathcal{C}_{X,m}) \wedge \mathcal{S}(\mathcal{C}_{Y,n})$

As an example, let us consider again the feature definitions of `Call Waiting` (defined in section 4.2.2) and `Incoming Call Screening` (defined in section 4.1.3). Let us also consider that sending a busy indication and denying a call presents a contradiction if applied to the same user and let us denote this using the contradiction relation:

- $\mathcal{K}$(busy_ind(B, A), deny_call(B, A)) – User B cannot send a busy indication to A and deny the call from A at the same time.

Performing pair-wise combination over the two feature descriptions $\mathcal{D}_{cw,3}$ and $\mathcal{D}_{ics,1}$, we observe that $|\mathcal{D}_{cw,3} \bullet \mathcal{D}_{ics,1}|$ presents an incoherence since it is possible to find a binding of variables that satisfies the incoherence rule. Let us consider that the system contains 4 possible users: *Alice*, *Bob*, *Carol*, *Dave*, and let us bind variables of $\mathcal{D}_{cw,3}$ and $\mathcal{D}_{ics,1}$ to users, such that their constraints are satisfied:

- $\mathcal{D}_{cw,3}$ – $A \leftarrow$ alice, $B \leftarrow$ bob, $C \leftarrow$ carol, $D \leftarrow$ dave.

$$\begin{aligned}
\mathcal{P}_{cw,3} &= \{\ \text{subs(alice, cw), concerns(carol, cw), busy(alice),} \\
&\qquad \text{talk(alice, bob), hold(alice, carol)}\ \} \\
\mathcal{T}_{cw,3} &= \{\ \text{call(dave, carol)}\ \} \\
\mathcal{R}_{cw,3} &= \{\ \text{busy_ind(carol, dave)}\ \} \\
\mathcal{C}_{cw,3} &= \{\ (\text{alice} \neq \text{bob}),\ (\text{alice} \neq \text{carol}),\ (\text{alice} \neq \text{dave}),\ (\text{bob} \neq \text{carol}), \\
&\qquad (\text{bob} \neq \text{dave}),\ (\text{carol} \neq \text{dave})\}
\end{aligned}$$

- $\mathcal{D}_{ics,1}$ – $A \leftarrow$ dave, $B \leftarrow$ carol, $C \leftarrow$ alice.

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

The rule is satisfied since:

1. (subs(alice, cw) $\in \mathcal{P}_{cw,3} \wedge$ concerns(carol, cw) $\in \mathcal{P}_{ics,1}$) $\wedge$
   (subs(carol, ics) $\in \mathcal{P}_{cw,3} \wedge$ concerns(carol, ics) $\in \mathcal{P}_{ics,1}$)
2. $\nexists\ \pi_1 \in \mathcal{P}_{cw,3},\ \pi_2 \in \mathcal{P}_{ics,1} \mid \mathcal{K}(\pi_1, \pi_2)$
3. $\mathcal{T}_{cw,3} = \mathcal{T}_{ics,1}$
4. $\exists\ \pi_3 \in \mathcal{R}_{cw,3},\ \pi_4 \in \mathcal{R}_{ics,1} \mid \mathcal{K}(\pi_3, \pi_4)$
5. $\mathcal{S}(\mathcal{C}_{cw,3}) \wedge \mathcal{S}(\mathcal{C}_{ics,1})$

Features have different subscribers but concern the same user. Carol is busy and held by Alice. When Carol receives a call (call(dave, carol)), `Call Waiting`, on Alice's side, informs that she should send a busy indication to the caller (busy_ind(carol, dave)), while `Incoming Call Screening`, on her side, informs that the call should be denied (deny_call(carol, dave)). We face a contradiction. Thus, we can conclude that $\mathcal{D}_{cw,3}$ and $\mathcal{D}_{ics,1}$ present an incoherence characterized by a contradiction.

4.2.3 Transitive Incoherence Rules

The word *transitive* refers to transitivity with respect to features. Namely, the two features could be $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$, subscribed by a different or the same user, where the results of $\mathcal{D}_{X,m}$ trigger $\mathcal{D}_{Y,n}$. We identify two possible incoherences for this situation: contradictions, when the results of the two descriptions present a contradiction, and loops, when the results of a description trigger the other one and vice-versa. Contradictions are identified by transitive incoherence rule #t1 and loops by transitive incoherence rule #t2.

The transitivity relation is an implication relation where $\mathcal{D}_{X,m}$ *implies* $\mathcal{D}_{Y,n}$, which can be denoted $\mathcal{D}_{X,m} \Rightarrow \mathcal{D}_{Y,n}$ or also $\mathcal{R}_{X,m} \Rightarrow \mathcal{T}_{Y,n}$.

Transitive Incoherence Rule #t1

Transitive rule #t1 identifies incoherences caused by the results of a feature triggering another feature that has results presenting a contradiction with results of the first feature. As usual, we consider two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$. An incoherence is present if, from Υ, the set of all possible users, it is possible to find a binding of property variables such that the combination of the two features present the following characteristics:

1. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ do not present a contradiction,
2. $\mathcal{R}_{X,m} \Rightarrow \mathcal{T}_{Y,n}$ ($\mathcal{R}_{X,m}$ is a superset of $\mathcal{T}_{Y,n}$, or, $\mathcal{R}_{X,m} = \mathcal{T}_{Y,n}$)

3. $\mathcal{R}_{X,m}$ and $\mathcal{R}_{Y,n}$ present a contradiction,
4. $\mathcal{S}(\mathcal{C}_{X,m})$ and $\mathcal{S}(\mathcal{C}_{Y,n})$, constraints of both features are satisfied.

The rule is formally defined as follows:

Variables of properties are bound using values provided by Υ such that:

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 a new feature `Outgoing Call Screening` and combine it with the feature `Call Forward on Busy` defined in section 4.2.2. `Outgoing Call Screening` allows the subscriber to establish a screening list and block the calls made to the users that are in this list. The list of properties corresponding to the requirements of `Outgoing Call Screening` is the following:

- `call(A, B)` – Call attempt from user A to user B.
- `ocs_list(A)` – User A is in the screening list of subscriber.
- `block_call(A, B)` – User A blocks outgoing call to user B.
- `call_blocked(A, B)` – User B is notified that his call has been blocked by user A.

Also, we must state that a call cannot be performed and blocked at the same time:

- $\mathcal{K}$(call(A, B), block_call(A, B)) – User A cannot perform a call to user B and block it at the same time.

Using the acronym `ocs` to denote `Outgoing Call Screening`, and number 1 for the single description part, the formal description of feature $\mathcal{D}_{ocs,1}$ is the following:

- $\mathcal{D}_{ocs,1}$ *Outgoing Call Screening, part 1*:

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

Performing pair-wise combination over the two feature descriptions $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ocs,1}$, we observe that $|\mathcal{D}_{cfb,1} \bullet \mathcal{D}_{ocs,1}|$ presents an incoherence since it is possible to find a binding of variables that satisfies the rule. Let us consider a system containing 3 possible users: *Alice*, *Bob*, *Carol*, and let us find a combination of those users to bind property variables of $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ocs,1}$ such that their respective constraints are satisfied, and rule #t1 is also satisfied:

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

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

- $\mathcal{D}_{ocs,1}$ – $A \leftarrow$ carol, $B \leftarrow$ alice.

$$\begin{aligned}
\mathcal{P}_{ocs,1} &= \{ \text{ subs(alice, ocs), concerns(alice, ocs), ocs_list(carol) } \} \\
\mathcal{T}_{ocs,1} &= \{ \text{ call(alice, carol) } \} \\
\mathcal{R}_{ocs,1} &= \{ \text{ block_call(alice, carol), call_blocked(alice, carol) } \} \\
\mathcal{C}_{ocs,1} &= \{ \text{ (carol} \neq \text{alice) } \}
\end{aligned}$$

The rule is satisfied since:

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

If Alice calls Bob (call(alice, bob)), `Call Forward on Busy` is activated and Bob forwards Alice to carol (redirected(alice, carol)). Alice calls Carol (call(alice, carol)) and `Outgoing Call Screening` blocks the call (block_call(alice, carol)). Bob's intention is that Alice calls Carol and Alice's intention is to block calls going to Carol.

Transitive Incoherence Rule #t2

Transitive incoherence rule #t2 identifies incoherences 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. The fact that their results present a contradiction or not is not checked since the goal is to identify loops caused by transitivity of features, not contradictions, already identified by the rule #t1. Given two feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$, an incoherence is present if, from Υ, the set of all possible users, it is possible to find a binding of property variables such that the combination of the two features presents the following characteristics:

1. $\mathcal{P}_{X,m}$ and $\mathcal{P}_{Y,n}$ do not present a contradiction,
2. $\mathcal{R}_{X,m} \Rightarrow \mathcal{T}_{Y,n}$ ($\mathcal{R}_{X,m}$ is a superset of $\mathcal{T}_{Y,n}$, or, $\mathcal{R}_{X,m} = \mathcal{T}_{Y,n}$)
3. $\mathcal{R}_{Y,n} \Rightarrow \mathcal{T}_{X,m}$ ($\mathcal{R}_{Y,n}$ is a superset of $\mathcal{T}_{X,m}$, or, $\mathcal{R}_{Y,n} = \mathcal{T}_{X,m}$),
4. $\mathcal{S}(\mathcal{C}_{X,m})$ and $\mathcal{S}(\mathcal{C}_{Y,n})$, constraints of both features are satisfied.

The rule is formally defined as follows:

Variables of properties are bound using values provided by Υ such that:

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. $\mathcal{R}_{Y,n} \supseteq \mathcal{T}_{X,m}$
4. $\mathcal{S}(\mathcal{C}_{X,m}) \wedge \mathcal{S}(\mathcal{C}_{Y,n})$

As an example, let us consider the feature definition of `Call Forward on Busy` given in section 4.2.2. Let us consider two different instances of the same feature description $\mathcal{D}_{cfb,1}$. Let us differentiate them using $\mathcal{D}_{\alpha-cfb,1}$ for the first instance and $\mathcal{D}_{\beta-cfb,1}$ for the second one, such that their composition is denoted $|\mathcal{D}_{\alpha-cfb,1} \bullet \mathcal{D}_{\beta-cfb,1}|$.

Performing pair-wise combination on them, $|\mathcal{D}_{\alpha-cfb,1} \bullet \mathcal{D}_{\beta-cfb,1}|$ presents an incoherence since it is possible to find a binding of variables that satisfies transitive incoherence rule #t2. Let us consider that the system contains 3 possible users: *Alice*, *Bob*, *Carol*, and let us bind variables of the two instances of $\mathcal{D}_{cfb,1}$ to users, such that their constraints are satisfied:

- $\mathcal{D}_{\alpha-cfb,1}$ – $A \leftarrow$ alice, $B \leftarrow$ bob, $C \leftarrow$ carol.
  $\mathcal{P}_{\alpha-cfb,1}$ = { subs(bob, cw), concerns(bob, cw), cfb(carol), busy(bob)}
  $\mathcal{T}_{\alpha-cfb,1}$ = { call(alice, bob) }
  $\mathcal{R}_{\alpha-cfb,1}$ = { redirected(alice, carol), call(alice, carol) }
  $\mathcal{C}_{\alpha-cfb,1}$ = { (alice $\neq$ bob), (alice $\neq$ carol), (bob $\neq$ carol) }
- $\mathcal{D}_{\beta-cfb,1}$ – $A \leftarrow$ alice, $B \leftarrow$ carol, $C \leftarrow$ bob.
  $\mathcal{P}_{\beta-cfb,1}$ = { subs(carol, cfb), concerns(carol, cfb), cfb(bob), busy(carol) }
  $\mathcal{T}_{\beta-cfb,1}$ = { call(alice, carol) }
  $\mathcal{R}_{\beta-cfb,1}$ = { redirected(alice, bob), call(alice, bob) }
  $\mathcal{C}_{\beta-cfb,1}$ = { (alice $\neq$ carol), (alice $\neq$ bob), (carol $\neq$ bob) }

The rule is satisfied since:

1. $\nexists\ \pi_1 \in \mathcal{P}_{\alpha-cfb,1},\, \pi_2 \in \mathcal{P}_{\beta-cfb,1} \mid \mathcal{K}(\pi_1,\, \pi_2)$
2. $\mathcal{R}_{\alpha-cfb,1} \supseteq \mathcal{T}_{\beta-cfb,1}$
3. $\mathcal{R}_{\beta-cfb,1} \supseteq \mathcal{T}_{\alpha-cfb,1}$
4. $\mathcal{S}(\mathcal{C}_{\alpha-cfb,1}) \wedge \mathcal{S}(\mathcal{C}_{\beta-cfb,1})$

The subscriber of each instance forwards its incoming calls to the subscriber of the other instance. The trigger of $\mathcal{D}_{\alpha-cfb,1}$ is call(alice, bob). Among the results of $\mathcal{D}_{\alpha-cfb,1}$ we have call(alice, carol). This is the trigger of $\mathcal{D}_{\beta-cfb,1}$ that has among its results call(alice, bob), trigger of $\mathcal{D}_{\alpha-cfb,1}$. As soon as $\mathcal{D}_{\alpha-cfb,1}$ is triggered (call(alice, bob)), its results trigger $\mathcal{D}_{\beta-cfb,1}$ (call(alice, carol)). The results of $\mathcal{D}_{\beta-cfb,1}$ then trigger $\mathcal{D}_{\alpha-cfb,1}$ (call(alice, bob)), and the system enters a loop.

We can then conclude that two different instances α and β of the feature description $\mathcal{D}_{cfb,1}$ can present a loop. This particular incoherence can be avoided by the use of a counter that is bounded to a certain number of iterations. Hence, a user does not forward the same call more than once.

4.3 Priorities

Most of the incoherences are due to the fact that two features try to influence the system in two different ways at the same time. Most of these incoherences can be avoided by using a priority mechanism. Priorities can be represented by the use of contradiction pairs. However, this technique has a limitation, presented in section 4.3.2.

4.3.1 Representing Priorities Using Contradiction Pairs

Let us define a property `priority(P)` where P denotes the priority of a feature. Let us state that two different priorities present a contradiction using the contradiction $\mathcal{K}$(priority(P), priority(Q)) if and only if P $\neq$ Q. By expressing that two features descriptions having different priorities present a contradiction, we state that only features having the same priorities can possibly present an incoherence.

For instance, to express that Call Forward Always has priority over Call Forward on Busy, we give them different priorities. The pre-conditions of the two feature descriptions respectively become:

$\mathcal{P}_{cfa,1}$ = { subs(B, cfb), concerns(B, cfb), cfb(C), priority(0) }

$\mathcal{P}_{cfb,1}$ = { subs(B, cfb), concerns(B, cfb), cfb(C), busy(B), priority(1) }

Due to the contradiction pair $\mathcal{K}$(priority(P), priority(Q)) if and only if P $\neq$ Q, $\mathcal{P}_{cfa,1}$ and $\mathcal{P}_{cfb,1}$ will present a contradiction since their priorities are different and thus, no incoherence will be identified. Moreover, the priorities stated in pre-conditions of each feature provides information to the designer.

4.3.2 Limitation & Possible Solution

The representation used in section 4.3.1 is not sufficient to model complex priority relations such as a feature having priority only over certain features and not over some others.

Let us consider three feature descriptions $\mathcal{D}_{X,1}$, $\mathcal{D}_{Y,1}$ and $\mathcal{D}_{Z,1}$, respectively denoted X, Y and Z to simplify the reading. Let us reason about assigning them priorities:

1. X has priority over Y. This can be modeled by the fact that priority(0) is assigned to X and priority(1) is assigned to Y.

2. Y has priority over Z. This can be modeled by keeping the priority previously assigned to Y (priority(1)) and assigning the priority(2) to Z.

3. X and Z have the same level of priority. This implies to modify the priority assigned to one of the two descriptions. Unfortunately, changing such priority invalidates one on the priorities relations defined in steps 1 and 2.

Hence, the three priority relations we reasoned about cannot be modeled at the same time. This is a limitation in the sense that a designer could not express such relations.

This problem can be solved by adding a property to the pre-conditions of feature descriptions: `priority(Features)`, where the variable $Features$ is a set of feature description names containing the names of the feature descriptions, over which the feature description considered has priority.

Such a solution implies to modify incoherences rules such that $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$ may present an incoherence if, and only if the priorities of their pre-conditions present an incoherence:

- Both descriptions have priority over each other; $\mathcal{D}_{X,m}$ has priority over $\mathcal{D}_{Y,n}$ and $\mathcal{D}_{Y,n}$ has priority over $\mathcal{D}_{X,m}$.
- None of the description has priority over the other one; $\mathcal{D}_{X,m}$ does not have priority over $\mathcal{D}_{Y,n}$ and $\mathcal{D}_{Y,n}$ does not have priority over $\mathcal{D}_{X,m}$.

4.4 In Summary

We have presented six rules that allow to identify incoherences between features at the requirements level and thus to identify feature interactions that can be present in a specification corresponding to the requirements. The rules are formally defined. As mentioned in section 4.2.1, variables are bound to users. The use of a finite number of values (users) insures a finite number of possible binding combinations for variables. Hence, the identification of incoherences is a decidable problem. Thus, such identification can be automated by the use of a logic predicate language. Chapter 5 presents our implementation of these rules as a tool for automatic filtering of incoherences and feature interaction detection.

Chapter 5

A Tool for Feature Interaction Detection

Test scenarios can be used for partial validation of systems, in the sense that whether a system can or cannot execute a scenario can reveal or exclude the existence of a particular error (although not necessarily all). The method proposed for filtering incoherences considers pairs of features. Given two features presenting an incoherence, the corresponding interaction can be detected in the specification using validation scenarios. The rules defined in chapter 4 allow manual identification of incoherences as well as manual test derivation to test a specification. However, automatic incoherences identification and automatic test suite generation for corresponding feature interactions detection are preferred.

Based on the rules presented in chapter 4, a tool for the automatic filtering of incoherences and detection of feature interactions is proposed. Given feature representations, the tool analyzes incoherences and generates reports. Based on these reports, validation test suites can be automatically obtained, thus, the existence of the corresponding interactions in the specification can be verified. The tool we propose is an application implementing two behaviors: the filtering of incoherences and the derivation of test suites, corresponding to the incoherences found for feature interaction detection.

In this chapter, we first present the scenario based validation of the LOTOS specification as it was manually done at the beginning of the project (section 5.1). Then, we present the tool . We first present the implementation of the filtering process (section 5.2). Then, we present the automatic derivation of test suites and explain what needs to be provided for such generation (section 5.3).

5.1 Scenario Based Validation

LOTOS allows many types of sophisticated validation procedures. However, some of these can be impractical or time consuming on large specifications. We propose a pragmatic approach, that can be quickly adopted in an industrial environment. For this purpose, validation of the LOTOS specification is based on scenarios. Test cases relative to the behavior of the system are produced and applied against the specification. The results of the scenarios are interpreted and allow the user to determine whether the model is correct or not, with respect to the scenarios applied against the specification. This section gives an overview of the manual validation performed on the specification.

5.1.1 Testing Strategy

In this method, scenarios are manually extracted from the UCM model. Note that the latter is a scenario-based model to start with. This allows the tester to derive scenarios by simply following possible paths in the model. This works well for the derivation of basic system and individual feature scenarios because these are based on the structure and syntax of the specification.

The sequence of activities that composes a given path is extracted from the UCM. That sequence is translated into a LOTOS process. The resulting scenario can be used for testing the LOTOS specification of the system. The purpose of scenarios derived from the UCM model is twofold: to validate the LOTOS specification against the UCM requirements and to test the system. Furthermore, scenarios are derived with three types of tests in mind:

Basic system properties – consist in testing the basic properties that the system must possess (the basic call). The scenarios involve simple test cases such as calling another party which is idle and calling another party which is busy.

Individual features properties – focus on testing features taken individually. Each feature needs to be working properly by itself before being integrated with other features for detecting interactions.

Feature interactions – focus on the detection of *interfering* feature interactions. These test scenarios aim at revealing problems that a given combination of features might inflict upon users.

The derivation of scenarios for feature interaction testing is less direct than for basic system and individual feature properties. Feature interaction involves the properties of the features and their behavior relating to each other and to the system. Some features can interact with many while others can interact only with a few. The strategy used consists in analyzing the features and targeting those pairs of features that have the highest probability of interaction. This analysis requires a good knowledge of the system and of its features.

5.1.2 Testing Using LOLA

The testing strategy consists in deriving scenarios from the UCM model, then applying them to the specification. A test suite composed of twenty scenarios testing the basic and individual properties of the system and its features were manually produced. The pairs of features having the highest likelihood of showing interactions were identified.

Test scenarios are specified in terms of LOTOS behaviors and applied against the LOTOS specification of the system by means of tools such as LOLA *(LOtos LAboratory, Universidad Politécnica de Madrid)*. Under LOLA, test scenarios are expected to exist as individual LOTOS processes within the LOTOS specification. This involves embedding the LOTOS behavior of each test scenario into a process structure. Provided with a targeted *success* event, LOLA synchronizes the given test process with the specification and attempts to follow all possible execution paths. The resulting verdict is then one of the following:

- MUST PASS – when all execution paths lead to the targeted *success* event
- REJECT – when the targeted *success* event is never reached
- MAY PASS – otherwise.

LOLA may also be instructed on the kind of exploration to perform:

Full Exploration – consists in exploring all the possible paths that can be reached when synchronizing with a test scenario. All cases conforming to those of the test scenario are analyzed. The execution time depends on the number of possible cases.

Partial Exploration with Random Walk – consists of a partial exploration of paths. The paths executed are selected using heuristics [35]. This is useful with tests that would require long execution times because heuristics provide much faster results (few seconds to few minutes). Even if the coverage is not 100% complete, experience shows that heuristic testing detects most of the specification faults quickly.

Unique Exploration – consists in visiting a single successful execution path. This is a convenient way to import of export scenarios to other tools. Such a trace, corresponding to *one* possible path, can be converted from or into a Message Sequence Charts (MSC) that can be used by other tools like SDT.

5.1.3 Feature Interaction Testing

Feature interaction detection is an important goal of the testing process. But, as mentioned, before testing for interactions between features, it is essential to test the basic properties of the system as well as the individual properties of the features. Once the system has been validated, specific FI tests derived from the UCM model are applied against the LOTOS specification.

All test scenarios are applied against the LOTOS specification using LOLA. Most of the scenarios are analyzed within a few seconds. Some scenarios can take days due to the number of states generated.

Test scenarios can be executed in such a way that LOLA displays intermediate results every $\boldsymbol{n}$ states (where $\boldsymbol{n}$ is an arbitrary number specified by the tester). This enables the tester to monitor the running test process.

The execution of scenarios can be driven by a script to automatically run them, possibly concurrently on different machines.

As an example, let us consider the following scenario: the user **B** is registered to *Call Forward on Busy* and forwards the calls to user **C**. User **A** is registered to *Originating Call Screening* and screens calls from user **C**. Given that **B** is busy and that **A** calls **B**, then **A** should be forwarded to **C** (because of **B**'s **CFB**) but it should also get a dial-Tone (because of **A**'s **OCS**). The corresponding LOTOS process is then coded as follows:

```
process ScenarioFI_OCS_CF_2r2 [ USER_to_DE, DE_to_USER,
                                Init, scenarioFI_OCS_CF_2r2 ]: noexit:=
  let specificDB:Database =
```

```
        UF(userB, FD(cfb, fArg(2003, noArg), endFeatureSet),
        UF(2002, FD(cfb, fArg(2003, noArg), endFeatureSet),
        UF(userA, FD(ocs, fArg(2003, noArg), endFeatureSet),
        UF(2001, FD(ocs, fArg(2003, noArg), endFeatureSet), emptyDB)))) in (
    Init !specificDB;

  USER_to_DE !f1 !userB !debB0 !offHook;         (* B goes offHook *)
  DE_to_USER !f2 !debB0 !userB !dialTone;        (* B gets dialTone *)
  USER_to_DE !f1 !userA !debA0 !offHook;         (* A goes offHook *)
  DE_to_USER !f2 !debA0 !userA !dialTone;        (* A gets dialTone *)
  USER_to_DE !f0 !userA !debA0 !dial !2002;      (* A dials B *)
  DE_to_USER !f2 !debA0 !userA !callInProgress;  (* A gets callInProgress *)
  DE_to_USER !f2 !debA0 !userA !dialTone;        (* A gets dialTone *)
  USER_to_DE !f1 !userA !debA0 !onHook;          (* A goes onHook *)
  DE_to_USER !f2 !debA0 !userA !toneOff;         (* A gets toneOff *)

  scenarioFI_OCS_CF_2r2; stop                    (* success *)
)
endproc
```

The output of the execution, with LOLA, of this test scenario on the specification is:

```
Analyzed states = 863
Generated transitions = 1121
Duplicated states = 0
Deadlocks = 0

Process Test = scenariofi_ocs_cf_2r2
Test result = MUST PASS.

successes = 259
stops = 0
exits = 0
cuts by depth = 0
```

This output provides information about the number of states of the graph generated, the number of generated transitions, and the number of successes. The number of successes equals the number of paths executed when the verdict is MUST PASS.

The first validation of the LOTOS specification has been done with scenarios that were derived by hand from UCM. This can lead to different problems. Not enough scenarios are produced or some important cases are missing. The validation is not complete. On the other hand, too many scenarios can be produced, which can lead to redundancies. Moreover, deriving all possible test cases (brute force) is not always a good choice, considering the processing time and the state explosion problem.

A solution is provided by our tool. Incoherences can be identified at the requirements level by using the filtering rules proposed in chapter 4. This provides useful information for testing the system. Later, this information is used to derive test scenarios in order to validate the specification. Thus, the specification is derived with knowledge of feature combinations presenting incoherences and, later, test scenarios allow to insure that these combinations do not lead to problems and that the specification is free of interactions.

5.2 Automatic Incoherences Filtering & FI Detection

This section presents the implementation of our filtering method for the automatic identification of incoherences. Given a set of feature descriptions, it is possible to identify incoherences manually by applying each rule to each possible pair. Thus, for n feature descriptions, n^2 pairs must be considered. Feature descriptions and rules are formally defined and based on sets of properties. Given a specific rule, the strategy followed for the identification of an incoherence is to bind the variables of properties to a proper combination of users such that the two features satisfy the rule. Such a task can require a significant amount of time, even for a small number of features.

Such a task can be automated by the use of a logic programming language. Representing rules and feature descriptions as logic predicates, the identification of an incoherence can be conducted using a logic programming interpreter. Such an interpreter tries to satisfy the predicates, and thus, tries to bind variables to all combinations until a combination that satisfies the predicates is found or all combinations of users have been tried. Logic programming languages such as Prolog [36, 37] or others can be used. Prolog is the language chosen. It has been developed at the University of Marseilles (France) [36] as a practical tool for programming in logic. The Prolog interpreter chosen is SWI-Prolog [41].

5.2.1 Implementation of Feature Descriptions & Rules into Prolog

Rules are translated into Prolog using the following mapping:

- Variables remain variables.
- Properties are represented as facts. Their syntax is the same as the one used for facts in Prolog.
- Sets of properties are represented as ordered lists of facts.
- A feature description is represented as a predicate, *feature*. The head of the predicate contains four lists: the name, pre-conditions, triggering events and results. The body contains the constraints, stated as relations between variables.

Let us consider the definition of Call Forward on Busy:

- $\mathcal{D}_{cfb,1}$ *Call Forward on Busy, part 1*:

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

The translation into Prolog is straightforward. The signature of the predicate *feature* is `feature(V, W, X, Y) :- Z`. Where variables V, W, X, Y respectively represent the lists containing name, pre-conditions, triggering events and results, and Z represents the body containing the restrictions.

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

In order to be able to bind variables to proper values (i.e. user names), users must be defined and the fact that variables must be bound to users must be specified explicitly. If not, the tool will still work but the Prolog interpreter will bind variables to internal default values. These are usually composed of an underscore, followed by a letter and numbers such as `_A340`. Even if still valid, such values can be difficult to interpret for the designer. It is easier to read `call(alice, bob)` than `call(_B747, _A340)`.

Users can be defined using a fact *user* containing the name of a user. Thus, defining four users for the system is done by defining four *user* facts, each one representing a different user: `user(alice).`, `user(bob).`, `user(carol).`, `user(dave).` In Prolog, the body of a feature definition contains a call to the *user* facts in order to bind variables to users. The complete definition of a feature description is then:

```
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 is called, Prolog binds variables A, B and C to different values taken from the *user* fact in such a way that relations between variables (constraints) are satisfied.

Contradictions between properties must be also clearly identified. If they are not, the tool will still work but some incoherences will not be identified. We use the predicate *contradiction_pair* to state a contradiction between two properties (facts). An example of a contradiction pair is that a user X cannot be busy and idle at the same time. This is expressed by the following Prolog fact:

```
contradiction_pair(busy(X), idle(X)).
```

A contradiction exists between two sets if it is possible to find two elements, one from each set, presenting a contradiction. The predicate *contradiction* is used to define possible contradictions between two facts. The code of the *contradiction* predicate follows:

```
contradiction(Set1, Set2):-
```

```
            member(X, Set1),
            member(Y, Set2),
            (contradiction_pair(X, Y) ; contradiction_pair(Y, X)), !.
```

As for feature descriptions, rules are translated into Prolog. Rules are implemented in the form of a predicate called *fi_check*. The signature of the predicate is the following: `fi_check(R, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2)`.

The variables of this predicate respectively denote: the name of the rule, the names (split in two parts) of the two features to be considered, the pre-conditions, triggering events and results of the first feature, and the pre-conditions, triggering events and results of the second one. The Prolog definition of rule #d1, given in chapter 4, section 4.2.2, is the following:

```
fi_check(d1, [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 always tries to satisfy a predicate. It contains a mechanism that allows the self-binding of variables in order to try to satisfy predicates and facts. For instance, if a query composed the fact *user* is called with a fixed value: `user(alice)`, the Prolog interpreter simply checks if a definition exists for such a fact and returns `true` or `false` whether such a fact exists or not. If the fact is called with a variable instead of a given value, the Prolog interpreter looks into the list of users and returns the next value. If this value does not satisfy the predicate, Prolog fetches the next one. And so on until the predicate is satisfied or no more values are available. Three kinds of queries can be made with the predicate *fi_check*:

- Variables are bound to values: the interpreter simply checks if the predicates listed in *fi_check* are satisfied on those values and returns `true` or `false` depending on whether the predicates are satisfied or not.

- Variables are unbound: the interpreter tries to bind variables to values that satisfy the predicate.

- Variables are partially bound: the interpreter tries to bind variables not bound to values that satisfy the predicate with respect to the values already given.

Thus, given two feature descriptions, a single query to one of the rules allows to find all incoherences that can be identified by this rule. Section 5.2.2 describes the application of the tool with respect to the identification of incoherences from the given descriptions.

5.2.2 Incoherences Filtering

Feature descriptions are contained into a text file. When the tool is invoked, the name of this file is given as a command line argument. The tool then elaborates all possible pairwise combinations and analyzes all pairs with each of the six rules to check whether an incoherence is present or not. In order not to report the same incoherence twice, the tool builds an internal database of incoherences found. In addition, each time an incoherence is found, a report is generated. The report contains a detailed description of the rule used and the incoherence found, and contains the incoherence itself.

An incoherence is represented by a fact called `fi` containing the description of the features involved. This fact is composed of nine parameters (1 to 9): the rule used (1), names of the features (2,3) and the pre-conditions (4,7), triggering events (5,8), and results (6,9) of both features. As an example, incoherence between $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ (presented in chapter 4, section 4.2.2) follows. The predicate has been reformatted to improve its readability. Text parts preceded by a % are comments.

```
fi(
    d1,                                                                 % (1)
    [cfb, 1],                                                           % (2)
    [ics, 1],                                                           % (3)
    [subs(bob, cfb), concerns(bob, cfb), cfb(carol), busy(bob)],        % (4)
    [call(alice, bob)],                                                 % (5)
    [redirected(alice, carol), call(alice, carol)],                     % (6)
    [subs(bob, ics), concerns(bob, ics), ics_list(alice)],              % (7)
    [call(alice, bob)],                                                 % (8)
    [deny_call(bob, alice), call_denied(bob, alice)]                    % (9)
).
```

Each report is composed of two parts: detailed information about the incoherence and the incoherence itself (`fi` fact). Detailed information 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 the complete information about the incoherence.

The first part, presenting the information about the incoherence, is written as Prolog comments. The second part is written as a Prolog `fi` fact, similar to the formatted one previously presented. Hence, a file containing reports is a Prolog database containing already known incoherences recorded as `fi` facts as well as user's readable information written as comments, thus ignored by the interpreter.

As an example, the report obtained for the incoherence between $\mathcal{D}_{cfb,1}$ and $\mathcal{D}_{ics,1}$ identified by rule #d1 follows. The report is shown in its initial form, with comments describing the incoherence. The `fi` fact following the description, already presented in reformatted form, is not shown here.

```
%-----------------------------------------------------------------------
% * Rule #d1 -> [cfb, 1] & [ics, 1]
%   Same user subscribed to two features having the
%   same triggering events and different results
```

```
%      + Features pre-conditions

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

%         - Pre-conditions of [ics, 1]
%             subs(bob, ics)
%             concerns(bob, ics)
%             ics_list(alice)

%      + Same triggering events

%          call(alice, bob)

%      + Different results

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

%         - Resulting events of [ics, 1]
%             deny_call(bob, alice)
%             call_denied(bob, alice)
%----------------------------------------------------------------------------
```

5.2.3 Filtering Algorithm

Considering $\mathcal{S}$, the set of all features of a specification, the list of all possible pair-wise combinations is obtained by a binary relation (*one to one* pairs) on $\mathcal{S} \times \mathcal{S}$. 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 are obtained by $\mathcal{U}$, basically, $\mathcal{S} \cup \mathcal{T} \times \mathcal{S} \cup \mathcal{T}$, which means that pair-wise combinations already analyzed have to be considered again, while they need not be since their results are already known. In order to avoid that, our tool implements two solutions:

The first solution consists in giving the set of already known incoherences such that for a given rule, only pairs not known as presenting an incoherence are considered. As we previously mentioned, the data in the report is a database of known incoherences. Thus, the set of already known incoherences can be obtained by loading a previous report into the tool. However, this only gives information about known incoherences. 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 $\mathcal{S} \times \mathcal{T}$. To distinguish them, we respectively call them *left hand set* and *right hand set*. 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. *Left hand set* and *right hand set* features are respectively represented using lists of Prolog facts called `lfeature` and `rfeature`.

Considering 3 feature descriptions $\mathcal{D}_{cfb,1}$, $\mathcal{D}_{ics,1}$, $\mathcal{D}_{ocs,1}$ already analyzed, and a new feature description $\mathcal{D}_{cw,1}$, the sets $\mathcal{S} = \{\mathcal{D}_{cfb,1}, \mathcal{D}_{ics,1}, \mathcal{D}_{ocs,1}, \mathcal{D}_{cw,1}\}$ and $\mathcal{T} = \{\mathcal{D}_{cw,1}\}$ are represented using the following facts:

```
lfeature([cfb, 1]).
lfeature([ics, 1]).
lfeature([ocs, 1]).
lfeature([cw, 1]).

rfeature([cw, 1])
```

And, $\mathcal{S} \times \mathcal{T} = \{|\mathcal{D}_{cw,1} \bullet \mathcal{D}_{cfb,1}|, |\mathcal{D}_{cw,1} \bullet \mathcal{D}_{ocs,1}|, |\mathcal{D}_{cw,1} \bullet \mathcal{D}_{ics,1}|, |\mathcal{D}_{cw,1} \bullet \mathcal{D}_{cw,1}|\}$ is the set of new combinations that need to be analyzed by the tool.

The first solution implies that only already known interacting pairs are not considered; all other pairs, even those already analyzed, are considered again. The second solution avoids this by directly targeting new pairs.

However, depending on the size of the specification, reusing already known incoherences, 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 depending on the number of descriptions to analyze.

The invocation of the tool must be done with at least one command line argument: the name of the file containing the specification. Three options can be specified:

`-g <file>` – Name of the file containing *right hand* and *left hand* sets.

`-k <file>` – Name of the file containing the list of already known incoherences.

`-i <file>` – Name of the file to which the report about incoherences are written.

When invoked, the interpreter loads the needed files. If the options `-k` is not used, no file containing the right and left hand sets is given. With respect to the solution previously described, the tool then builds the two sets such that both of them contains all the descriptions, thus, all combinations are analyzed. Then, the tool looks for incoherences by applying each rule to each pair and generates a report for each incoherence found. The complete description of the procedure follows:

1. Load the features description file.
2. If a file containing left and right hand sets is given in the command line, load it. Else, build sets such that each one contains all features.
3. If a file containing already known incoherences is given in the command line, load it.
4. For each existing rule
   (a) For each pair in $\mathcal{S} \times \mathcal{T}$

i. Try all possible values for variables to satisfy the rule
ii. If an incoherence is found, update the internal database and generate a report. Else do nothing.

Prolog tries to satisfy predicates thus insuring that all possible values are tried. Finding a proper combination of users is part of the task automatically managed by the interpreter. Pair-wise combinations can also be automatically built by a single call to a predicate. The algorithm is split in two symmetric parts: one concerning pairs (s, t) and the other concerning pairs (t, s) where $s \in \mathcal{S}$ and $t \in \mathcal{T}$. The Prolog implementation of part 4 of the algorithm is given below:

```
% default predicate
fi_search_left_to_right:-
        rule(R),         % rule
        lfeature(F1),    % feature description from left hand set
        rfeature(F2),    % feature description from right hand set

        % no FI has been found yet for this combination with rule R
        not(fi(R, F1, F2, _, _, _, _, _, _)),

        % try to find a possible FI
        fi_check(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),

        % add FI to the internal database
        assert(fi(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2)),

        % generate report corresponding to the FI
        fi_report(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),

        % loop (recursive call)
        fi_search_left_to_right, !.

% predicate used if default one can not be satisfied (FI already found)
fi_search_left_to_right:-
        rule(R),
        lfeature(F1), rfeature(F2),
        (fi(R, F1, F2, _, _, _, _, _, _)), !.
```

5.3 Automatic Test Suite Generation

Incoherences identification is used as a filtering method. Since the information about features is obtained from the requirements, identifying an incoherence does not imply that the corresponding interaction exists in the specification. Nevertheless, a designer being aware of such an incoherence can take extra care in designing the system. Hence, this can lead to the avoidance of several interactions at the design stage.

The information provided by the reports can be used to derive validation test suites in order to verify that interactions corresponding to incoherences are not present in the specification. However, depending on the number of incoherences, manual test suite derivation can be time consuming. Considering that the specification follows the requirements, establishing mapping rules from properties (used to formalize feature descriptions) to scenario parts enables the automatic generation of scenarios, and thus, test suites.

5.3.1 From Properties to Scenario Parts

Properties used to describe a feature illustrate the state in which the feature must be, the event(s) triggering the feature and the results produced by the activation of this feature. Properties are obtained from the requirements. The LOTOS specification is also derived from the requirements. Depending on the language used, features are represented differently and thus, validation scenarios use different representations such as Message Sequence Charts in SDL or actions in LOTOS.

Given a LOTOS specification involving a pair of features for which an incoherence was identified according to the method described in section 5.2, a validation scenario is a sequence of actions, representing the behavior of a feature interaction. Each action represents a part of the behavior, such as the establishment of a call or a dial-tone. As properties describe the features, actions of a test scenario correspond to actions of the behavior described in requirements. It is possible to establish a direct mapping between properties and scenario parts. Then, given an incoherence, a scenario can be automatically derived by following the model presented in section 5.2.3 and using the mappings between properties and scenario parts.

Establishing mapping rules to go from a feature description to a LOTOS test scenario consists in identifying the LOTOS action(s) corresponding to each property of the set of feature descriptions. The mapping is done by associating each property to an action of the specification. To generate a test scenario, the algorithm uses five predicates for which signatures (names and parameters) are pre-defined. The body of the predicates needs to be programmed by the tester to implement scenario generation. The predicates are respectively:

test_header(*[F1, PndF1]***,** *[F2, PndF2]***,** *Result***,** *R***)** – This predicate builds the header of the test scenario. It expects the names of two features *F1* and *F2* and the lists of their pre-conditions: *PndF1* and *PndF2*. In addition, the predicate also expects the name of the feature in which the scenario results, *Result*, and the rule used, *R*.

test_pre_conditions(*List***)** – This predicate is translates a list of pre-conditions into a list of scenario parts (e.g. actions, in LOTOS). It expects a list of pre-conditions, decomposes it into single elements (properties) and then calls a predicate corresponding to each of them.

test_triggering_events(*List***)** – This predicate translates a list of triggering events into a list of scenario parts (e.g. actions, in LOTOS). It expects a list of triggering events, decomposes it into single elements (properties) and then calls a predicate corresponding to each one of them.

test_results(*List*) – This predicate translates a list of results into a list of scenario parts (e.g. actions, in LOTOS). It expects a list of results, decomposes it into single elements (properties) and then calls a predicate corresponding to each one of them.

test_footer(*List*) – This predicate generates the end of the test scenario. The end of the scenario does not contain any specific information. The parameter list is for future use.

The test generation algorithm is based on the five predicates presented above. The tester must provide the internal code of each one of them such that they can be used by the interpreter. The body of these predicates is defined depending on what needs to be provided for test generation. Additional predicates can be defined, such as those needed to map users to phone numbers.

A property may have different meanings depending on the set (pre-conditions, triggering events or results) they belong to. As an example, let us consider Call Forward, defined in chapter 4, section 4.2.2. The property `call` composed of two variables is present in both triggering events and results. However, the meaning of a call is not the same in the triggering events where it represents an incoming call than in the results where it represents an outgoing forwarded call. It is more convenient to use different sub-predicates to map properties from pre-conditions, triggering events and results. Pre-conditions are mapped using a predicate called *pre-condition*, triggering events are mapped using a predicate called *triggering_event* and results are mapped using a predicate *results*. Such a partitioning allows specify different mappings for the same property, depending on the set it belongs to in the feature description.

Pre-conditions, triggering events and results describe part of the behavior and are considered as ordered sets. Elements composing a results must appear in a certain order, defined in requirements. The use of Prolog lists for feature description insures that the order between properties is respected in the scenario generation algorithm. This gives the possibility to the testers and designers to insure that events occur in a specific order.

5.3.2 Test Suite Principles

Feature interaction validation of a specification assumes that the basic system properties have been validated and the features work properly in isolation. Nevertheless, the tool offers the automatic generation of single scenarios for single features validation. The behavior of a feature is contained in the feature description, thus, using the predicates presented section 5.3.1, a given feature description can be mapped into a single scenario corresponding to its behavior.

Applied against the specification, the scenario can be successful, meaning that the basic behavior of the feature is working, or, rejected (or not always successful), meaning that the feature is not working properly in isolation. Even if single feature testing is not the main goal of the tool, this offers the possibility of automatically insuring that the basic behavior of the features has been correctly specified.

Feature interaction testing scenarios are more complex. An interaction, corresponding to an incoherence identified in the filtering process, can be characterized by non determinism, a contradiction (which can be also seen as a non determinism) or a loop. We define a general model corresponding to each of the three possible kinds of interactions. The models defined are then used for the derivation of test scenarios and analysis of the results following their application on the specification using LOLA.

Non-determinism – Non-determinism implies that the result is ambiguous. The tool is not able to know which result is intended by the designer. Thus, interaction detection is done using a test suite composed of two scenarios. Both scenarios model a case 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 with LOLA is interpreted in the following manner:

- Both scenarios are rejected (REJECT or MAY PASS). This definitely indicates a problem. It tells the tester that in any case, neither feature behaves properly.
- Both scenarios are successful (MUST PASS) means that both results can occur. This indicates a problem in the specification because only one feature should be activated. However, depending on the features, this result can be acceptable. It is up to the tester to insure that the behavior is the one intended.
- One scenario is successful (MUST PASS) and the other is rejected (REJECT or MAY PASS). This indicates that, at least, no non-determinism exists. However, the behavior may not be the one intended. If the tester expects the rejected scenario to be successful and vice-versa, then a problem is present. As for the previous case, it is up to the tester to insure that the behavior is the one intended.

Contradictions – A Contradiction also presents non-determinism. The results of the two involved features are in contradiction, and, as for non-determinism, the result is not known. Thus, scenarios for this kind of interaction follow the same model as the one followed by the non-deterministic test suites.

Loops – The model followed by loop scenarios is different. A loop can either occur or not occur. Then, only one scenario is needed to detect such interaction. The scenario illustrates the case where the loop is occurring. Thus, rejection of the scenario indicates that no loop occurs. If the scenario is successful, then it can be concluded that the loop effectively occurs in the specification.

A tester having knowledge of the models used can interpret the results of the application of scenarios and give a verdict. Explanations concerning the derivation of scenarios follows.

5.3.3 Scenarios Generation

This section presents the implementation of scenario generation. It includes single feature validation scenarios and feature interaction validation scenarios. Examples, based on the Mitel specification, are given in chapter 6.

Single Feature Test Scenarios Generation

The goal of single feature test scenario generation is to translate the behavior contained in a feature description into a scenario. This is implemented as a predicate calling the five mandatory predicates previously defined in section 5.3.1. The Prolog predicate used is:

```
isolation_test(F1, PcnF1, TrgF1, ResF1):-
          test_header([F1, PcnF1]),
          test_pre_conditions(PcnF1, []),
          test_triggers(TrgF1),
          test_results(ResF1),
          test_footer.
```

Since only one feature is considered, a different predicate `test_header` is used. In addition, the predicate `test_pre_conditions` are called with an empty list as second argument instead of the pre-conditions of the second feature.

Feature Interaction Test Suites Generation

As explained in section 5.3.2, we identified three different kinds of interactions. Non-deterministic and contradictory interactions can be tested based on the non-determinism they present. They both need two scenarios per interaction. However, the kind of rule (direct incoherence rules or transitive incoherence rules) corresponding to them must be taken in account. Interactions presenting a loop only need one scenario.

All direct incoherence rules (#d1, #d2, #d3, #d4) identify non-determinism or contradictions. The corresponding validation test suites can be derived using the same model. Rule #t1 identifies contradictions but is a transitive rule, thus, a different model must be used. Rule #t2 identifies loops and, thus, also needs to use a different model. Hence, we distinguish three kind of test suites:

Direct test suites – Concerning rules #d1, #d2, #d3 and #d4, these test suites are composed of two test scenarios, respectively presenting the results of the first and the second feature. The predicate used is the following:

```
fi_test(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
        ((R = d1); (R = d2);(R = d3); (R = d4)),
        test_header([F1, PcnF1], [F2, PcnF2], F1, R),   % F1
        test_pre_conditions(PcnF1, PcnF2),
        test_triggers(TrgF1),
        test_results(ResF1),
        test_footer,

        test_header([F1, PcnF1], [F2, PcnF2], F2, R),   % F2
        test_pre_conditions(PcnF1, PcnF2),
        test_triggers(TrgF2),
        test_results(ResF2),
        test_footer.
```

Transitive test suites – Concerning rule #t1, these test suites are also composed of two test scenarios. One illustrates the case where the scenario stops after the first feature, and thus, the second one is not activated. The other illustrates the case where the second feature is activated. The Prolog predicate is:

```
fi_test(t1, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
        test_header([F1, PcnF1], [F2, PcnF2], F1, t1),   % F1
        test_pre_conditions(PcnF1, PcnF2),
        test_triggers(TrgF1),
        test_results(ResF1),
        test_footer,

        test_header([F1, PcnF1], [F2, PcnF2], F2, t1),   % F2
        test_pre_conditions(PcnF1, PcnF2),
        test_triggers(TrgF1),
        test_results(TrgF2),
        test_results(ResF2),
        test_footer.
```

Loop test suites – Concerning rule #t2, these test suites are composed of only one test scenario. The test scenario illustrates the activation of the first feature, followed by the activation of the second one, followed, again, by the activation of the first feature. The corresponding predicate is:

```
fi_test(t2, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, _, ResF2):-
        test_header([F1, PcnF1], [F2, PcnF2], ['a', 'loop'], t2),
        test_pre_conditions(PcnF1, PcnF2),
        test_triggers(TrgF1),
        test_results(ResF1),
        test_results(ResF2),
        test_results(ResF1),
        test_footer.
```

The production of validation test suites needs the file containing the feature descriptions and the file containing the mapping rules. A test suite is produced by calling the `fi_test` predicate relative to the rule identifying the incoherence corresponding to the interaction. Known incoherences can be optionally specified. In such a case, the corresponding test suites are generated; else, the filtering algorithm is applied and the corresponding test suites are generated. Four possible invocations are possible:

- Incoherences filtering,
- Test suite derivation for features in isolation,
- Test suite derivation for feature interactions,
- Both filtering and test suite generation combined together.

The complete Prolog implementation can be found in appendix B. The tasks are chosen at invocation, by the use of arguments. If invoked with no arguments (or wrong arguments) the tool prints its usage, as follows:

```
Usage: fi-lookup <command>
    * where command must be one of the following:

    -filtering <features> [options]
    --> find incoherences
        * where options include:
        -g <file> --- feature sets to combine
        -k <file> --- list of already known incoherences
        -i <file> --- file to which reports are written

    -testiso <features> <mapping-rules> [options]
    --> produces test suites for features in isolation
        * where options include:
        -t <file> --- file to which test suites are written

    -testfi <mapping-rules> <known incoherences> [options]
    --> produces test suites for given incoherences
        * where options include:
        -t <file> --- file to which test suites are written

    -filttest <features> <mapping-rules> [options]
    --> finds incoherences & produces corresponding test suites
        * where options include:
        -g <file> --- feature sets to combine
        -k <file> --- list of already known incoherences
        -i <file> --- file to which reports are written
        -t <file> --- file to which test suites are written
```

5.4 In Summary

We showed that the automation of incoherences filtering and test suite generation for feature interaction detection is possible using a logic programming language. The complete Prolog code of the tool is 850 lines long (commented). An evaluation, based on the Mitel specification, and on the Feature Interaction Workshop 2000 [4] is presented in chapter 6 with detailed examples of incoherences filtering. Test suite generation is also presented.

Chapter 6

Application & Evaluation

Our incoherences identification approach is based on a filtering process followed by a validation process that uses automatic generation of validation scenarios. This chapter presents the results obtained from the application of our method on two different specifications. The algorithmic complexity of the analysis as well as the usefulness of the method are presented. In addition, a comparison with another approach is discussed.

6.1 Application

We applied our approach to two different case studies: Mitel specification and the Feature Interaction Workshop 2000 [4] specification [42]. The application on the Mitel specification consisted in filtering incoherences, deriving test suites for the corresponding interactions and using these suites to test the specification. The application on the Feature Interaction Workshop consisted in applying the filtering process to identify incoherences.

6.1.1 Case Study 1 – Mitel

As mentioned in chapter 2 and chapter 3, the approach applied to the Mitel project consists in expressing requirements with a Use Case Maps model, 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. We first manually derived feature interaction test suites to validate the specification. Then, we built our tool and applied the filtering process and the derivation of test suites to automatically validate the LOTOS specification.

Part 1 – Filtering Process

Using Use Case Maps and additional documentation provided with the set of UCM by Mitel Corp., the features were modeled and our filtering method was applied to it. The 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 (ARC) and Time Reminder (TMR). These features are presented in chapter 3, section 3.2.2.

Some of the features are represented using a single description while some 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. They add up to twenty feature descriptions. As an example of features definitions, 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) 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) 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 contradiction 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_pair* in the following form:

- `contradiction_pair(call(A, B), call(A, C)):- B \= C`

Our filtering method identified a total of 43 incoherences. Eight of them concern incoherences involving Call Forward Always and Call Forward on Busy. Brief explanations, based on the reports given by the tool, are given here. The 43 reports can be found in appendix A.

Rule #d2 → [cfb, 1] & [cfa, 1] – Contradiction between Call Forward Always and Call Forward on Busy when some user subscribes to both features and the features forwards his calls to different users.

Rule #t1 → [cfb, 1] & [cfb, 1] – Contradiction between Call Forward on Busy and Call Forward on Busy when some user subscribes to Call Forward on Busy and forwards his calls to a user that also subscribes to Call Forward on Busy but forwards his calls to someone else.

Rule #t1 → [cfb, 1] & [cfa, 1] – Contradiction between Call Forward on Busy and Call Forward Always when some user subscribes to Call Forward on Busy and forwards his calls to a user that subscribes to Call Forward Always but forwards his calls to someone else.

Rule #t1 → [cfa, 1] & [cfb, 1] – Contradiction between Call Forward Always and Call Forward on Busy when some user subscribes to Call Forward Always and forwards his calls to a user that subscribes to Call Forward on Busy but forwards his calls to someone else.

Rule #t1 → [cfa, 1] & [cfa, 1] – Contradiction between Call Forward Always and Call Forward Always when some user subscribes to Call Forward Always and forwards his calls to a user that also subscribes to Call Forward Always but forwards his calls to someone else.

Rule #t2 → [cfb, 1] & [cfb, 1] – Loop between Call Forward on Busy and Call Forward on Busy when two users subscribes to Call Forward on Busy and each one forwards his calls to the other.

Rule #t2 → [cfb, 1] & [cfa, 1] – Loop between Call Forward on Busy and Call Forward Always when two users respectively subscribe to cfb and Call Forward Always, and each one forwards his calls to the other.

Rule #t2 → [cfa, 1] & [cfa, 1] – Loop between Call Forward Always and Call Forward Always when two users subscribe to Call Forward on Busy and each one forwards his calls to the other.

A detailed summary of the incoherences identified between all pairs of feature descriptions is presented in Table 6.1:

Incoherences									
	OCS	ICS	CFA	CFB	CT	CP	CW	ARC	TMR
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	
ARC	—	—	—	—	—	—	—		
TMR	—	—	—	—	—	—	—	—	

Number of incoherences per rule and type						
Rule	#d1	#d2	#d3	#d4	#t1	#t2
Number of incoherences	13	3	4	1	19	3
Ratio per rule	30%	7%	9.5%	2.5%	44%	7%
Ratio per type	49%				51%	

Table 6.1: Mitel Case Study: Incoherence Filtering Results

Spaces indicate that the tool did not identify any incoherence. A distinction is made between direct and transitive incoherences. In addition, we make a distinction between direct incoherences concerning features subscribed by the same user and those concerning features subscribed by different users. Such distinctions allow to classify incoherences into three groups, presented below.

Rules d1 & d2 – identify direct incoherences present between two features having the same subscriber and presenting incoherent results.

Rules d3 & d4 – identify direct incoherences present between two features having different subscribers and presenting incoherent results.

Rules t1 & t2 – identify transitive incoherences present between two features where one triggers the other and the results are incoherent or lead to a loop.

The results show that most of the incoherences identified are either direct incoherences present between two features subscribed by the same user (rules #d1 and #d2) or transitive incoherences (rules #t1 and #t2). Only a few direct incoherences are identified for pairs of features subscribed by different users (rules #d3 and #d4).

We interpret the results as follows: incoherences identified by rules #d1, #d2, #t1 and #t2 are easier to foresee. Incoherences identified by rules #d3 and #d4 are much more complex and thus, can be more difficult to avoid. Hence, having only a few incoherences identified by rules #d3 and #d4, as it is the case here, indicates that the designers have to deal with simple incoherences and thus, refining requirements or specification to avoid incoherences takes less time and effort.

Part 2 – Test Suites Generation

As explained in chapter 5, the tool produces test suites composed of one or two scenarios. Scenarios are produced using five predicates: *test_header*, *test_pre_conditions*, *test_triggers*, *test_results* and *test_footer*.

These predicates must be coded by the tester and must contain the mapping rules needed for the scenarios derivation. We modeled the mapping rules using the five conventional predicates as well as some additional ones when needed:

- **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 instructions. This predicate calls another sub-predicate `database_arg` to build the arguments of the database corresponding to the attributes of the features.

- **test_pre_conditions** – This predicate is used to decompose the list of pre-conditions properties and to produce the LOTOS event(s) corresponding 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 `test_pre_condition` 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 LOTOS line(s) corresponding to each and every property.

- **test_results** – This predicate is built on the same principles as the previous one and uses the sub-predicate `test_result` instead of `test_trigger`.

- **test_footer** – This predicate is simply used to produce the end of the LOTOS test process, which does not contain anything special.

In the LOTOS specification, a user is associated with a phone number and a DA (see chapter 3, section 3.2 for more details).

Moreover, users in the LOTOS specification have different names than the users of the Prolog features specification. Mapping between users and their phone numbers and DAs is done by using a predicate `lotos_user`. Examples of the predicates we defined are:

- Mapping rule used to build the test header:

```
test_header([[F1, Fx], CndF1], [[F2, Fy], CndF2], [F, X], R):-
        write('process ScenarioFI_R'), write(R), write('_'),
        write(F1), write('_'), write(Fx), write('_and_'),
        write(F2), write('_'), write(Fy), write('_resulting_in_'),
        write(F), write('_'), write(X), write('_['), nl,
        write('\tUSER_to_DE, DE_to_USER, Init, success ]: noexit:=\n'),
        write('\n  let specificDB:Database ='),
        lotos_database([[F1, CndF1], [F2, CndF2]]),
        write(' in ( \n\n    Init !specificDB;\n\n').
```

- Mapping rule used to decompose sets of pre-conditions:

```
test_pre_conditions(CndF1, CndF2):-
        test_pre_conditions(CndF1),
        test_pre_conditions(CndF2), !.

test_pre_conditions([]).

test_pre_conditions([Elem|List]):-
        test_pre_condition([Elem]),
        test_pre_conditions(List).
```

- Mapping rule used map between a user and his phone number and DA.

```
lotos_user(alice, 'userA', '2001', 'debA0').
lotos_user(bob, 'userB', '2002', 'debB0').
lotos_user(carol, 'userC', '2003', 'debC0').
lotos_user(dave, 'userD', '2004', 'debD0').
```

- Examples of mapping rules for the pre-condition property `busy(X)`:

```
test_pre_condition([busy(X)]) :-
        lotos_user(X, User, _, Deb),
        write('   USER_to_DE !'), write(User), write(' !'), write(Deb),
        write(' !offHook'), write(';\n'),
        write('   DE_to_USER !'), write(Deb), write(' !'), write(User),
        write(' !dialTone'), write(';\n').
```

Based on these mapping rules, our tool produces LOTOS test suites. Directly applied to the specification, these test suites allow us to perform feature interaction validation and to verify that no interaction can be found in the specification.

As an example, let us consider the incoherence identified by rule #d2 for Call Forward Always and Call Forward on Busy. This incoherence is characterized by the fact that if a user 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, an incoherence is present when someone calls this user when he is busy. This contradiction is due to both features being triggered and leading to contradictory results since calls are forwarded to different users. The report for this incoherence follows.

```
% * Rule #d2 -> [cfb, 1] & [cfa, 1]

%   Same 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 is composed 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 (resulting in Call Forward on Busy) is shown below.

```
process ScenarioFI_Rd2_cfb_1_and_cfa_1_resulting_in_cfb_1[
        USER_to_DE, DE_to_USER, Init, success ]: noexit:=

  (* database initialization *)
  let specificDB:Database =
        UF(userB, FD(cfb, fArg(2003), noArg), endFeatureSet),
        UF(2002, FD(cfb, fArg(2003), noArg), endFeatureSet),
        UF(userB, FD(cfa, fArg(2004), noArg), endFeatureSet),
        UF(2002, FD(cfa, fArg(2004), noArg), endFeatureSet),
        emptyDB)))) in (
    Init !specificDB;

    (* busy(bob) *)
    USER_to_DE !userB !debB0 !offHook;
    DE_to_USER !debB0 !userB !dialTone;

    (* call(alice, bob) *)
    USER_to_DE !userA !debA0 !offHook;
    DE_to_USER !debA0 !userA !dialTone;
    USER_to_DE !userA !debA0 !dial !2002;
    DE_to_USER !debA0 !userA !callInProgress;

    (* redirected(alice, carol), call(alice, carol) *)
    DE_to_USER !debA0 !userA !redirectTone;

    (* ring(alice, carol) *)
    DE_to_USER !debC0 !userC !ringingOn;

    success; stop
  )
endproc
```

The application of the two test scenarios using LOLA allows the tester to verify that:

- First, only one of the two features is always triggered: applied to the specification using LOLA (see chapter 5, section 5.1.2), 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 results (scenario resulting in a MUST PASS) and the requirements.

The Use Case Maps model and the LOTOS specification already existed for 6 of the features in consideration when we formalized our rules and built the tool. Hence, our method couldn't influence the Use Case Maps or the LOTOS models for these 6 features (OCS, ICS, CFA, CFB, CT, CP). However, we were able to identify incoherences for all the 9 features including some that were not identified by Mitel's designers for the three last features. All possible incoherences identified were incoherences that could possibly lead to interactions in the specification.

In addition, we derived test suites and tested the LOTOS specification for the six features already specified No feature interactions were found, due to the fact that feature studied were well known to the specifiers; interactions were avoided by Mitel designers at the Use Case Maps level.

6.1.2 Case 2 – The Feature Interaction Workshop 2000

Following the first feature interaction detection contest held in 1998 [3], a second feature interaction contest was held in May 2000 [4, 43], in conjunction with the sixth international workshop on feature interactions in telecommunications and software systems. The goal of the workshop was to *"provide a simple comparison of different automated tools for feature interaction detection"*. In this second contest, the basic call model and the features are specified using Finite State Machines [42].

Most of the features are composed of two parts: *subscriber*, describing the behavior relative to the subscriber, and *everyone*, describing the behavior relative to non-subscribers. Fig. 6.1 presents the specification of the Terminating Call Screening feature.

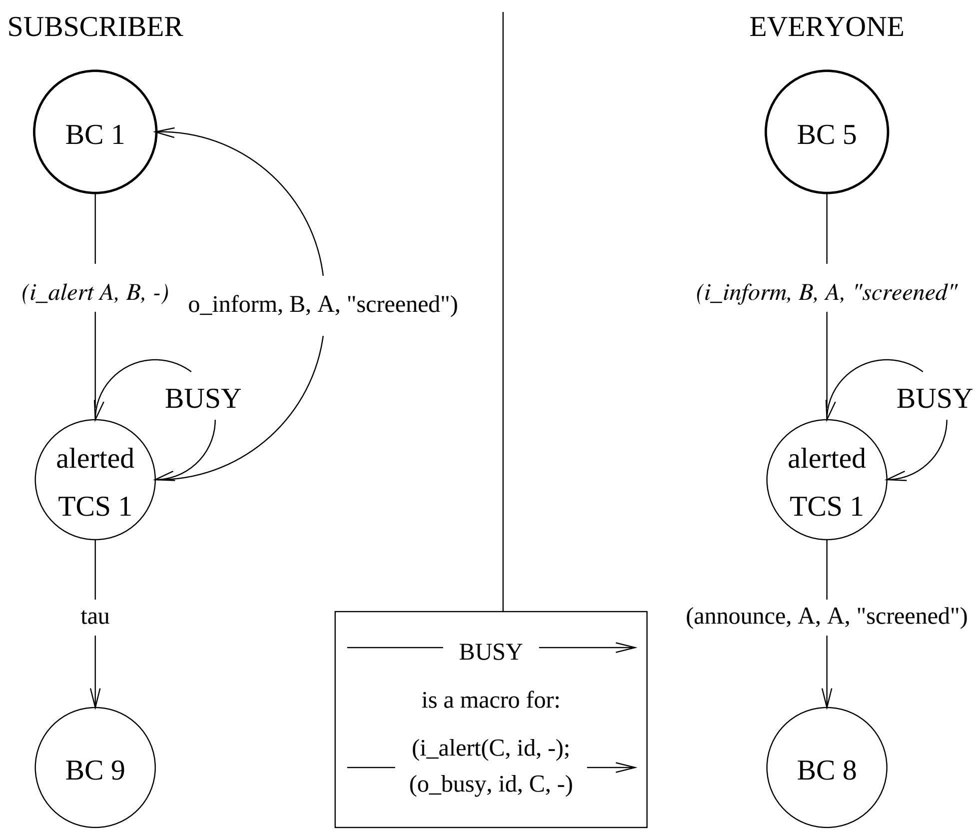

Figure 6.1: Terminating Call Screening

We only wanted to test our filtering method on a different set of features, so no test suites were derived. Moreover, we only applied our filtering method to the ten features of phase one of the contest. A formal specification of the features was already given (using Finite state machines); it was translated into our Prolog representation. The mapping of the FSM representation into Prolog feature descriptions is based on the following rules:

- The name of a description is composed of the name of the feature for the first part and the state of the feature description for the second part.
- States are mapped into pre-conditions. A list of pre-conditions includes the properties characterizing the state of the feature along with the name of the state.
- Transitions representing an incoming event are mapped into triggering events.
- Transitions representing an output are mapped into results.
- Only external observable actions, corresponding to inputs and outputs, are considered. Thus, states are only shown in pre-conditions; they do not appear in triggering events or results.
- An input followed by an output going from a state to another via an intermediary state (as is the case for *(i_alert A, B, -)* and *(o_inform B, A, "screened")* on fig. 6.1) are represented as a triggering event and a result. Intermediate states are states that do not correspond to any input or output. These states are not considered.

Our Prolog representation of Terminating Call Screening (fig. 6.1) is the following:

```
% *** Subscriber ***

% *** bc(1) ***
% B is subscriber, is idle and receives a call from A, A is screened
feature([tcs, bc(1)],
        [subs(C, tcs), concerns(C, tcs),             % pre-conditions
         tcs_list(B), idle(C), bc(1)],
        [alert(B, C, -)],                            % triggering events
        [inform(C, B, screened)]                     % results
    ):-
        user(B), user(C), C \= B.                    % constraints

% *** tcs(1) ***
% while screening a call, if a new call incomes, busy indication is sent
feature([tcs, tcs(1)],
        [subs(B, tcs), concerns(B, tcs), tcs_list(A),  % pre-conditions
         busy(B), alerted(B), tcs(1)],
        [alert(A, B, -)],                            % triggering events
        [busy(B, A, -)]                              % results
    ):-
        user(A), user(B), B \= A.                    % constraints
```

```
% *** Everyone ***

% *** bc(5) ***
% A calls B and is in screening list of B, A is screened by B
feature([tcs, bc(5)],
        [subs(B, tcs), concerns(A, tcs),                  % pre-conditions
         tcs_list(A), bc(5)],
        [inform(B, A, screened)],                         % triggering events
        [announce(A, A, screened)]                        % results
    ):-
        user(A), user(B), B \= A.                         % constraints

% *** tcs(2) ***
% while being screened, if a call is received, busy indication is sent
feature([tcs, tcs(2)],
        [subs(B, tcs), concerns(A, tcs), busy(A),         % pre-conditions
         tcs_list(A), annoucement(A), tcs(2)],
        [alert(C, A, -)],                                 % triggering events
        [busy(A, C, -)]                                   % results
    ):-
        user(A), user(B), B \= A,                         % constraints
        user(C), C \= A, C \= B.
```

The ten features were represented using 97 description parts. The application of our filtering

	CFB	TL	TCS	CW	TWC	RC	CNDB	RBWF	VM	SB
CFB	2	1	2	4	1	1		5	3	1
TL	—			1	7			6	3	
TCS	—	—		2				3		
CW	—	—	—		2	1		11	9	1
TWC	—	—	—	—		1	1	33	24	1
RC	—	—	—	—	—		1	1		1
CNDB	—	—	—	—	—	—				1
RWBF	—	—	—	—	—	—	—		16	1
VM	—	—	—	—	—	—	—	—		
SB	—	—	—	—	—	—	—	—	—	

—

Number of incoherences per rule and type						
Rule	#d1	#d2	#d3	#d4	#t1	#t2
Number of incoherences	94	1	49	0	2	1
Ratio per rule	64%	0.75%	33%	0%	1.5%	0.75%
Ratio per type	97.75%				2.25%	

Table 6.2: Feature Interaction Workshop Case Study: Results

method on the whole specification reported 149 incoherences. Table 6.2 summarizes the incoherences identified. Most of them were identified by direct incoherence rules.

The number of incoherences is significant but some of them are incoherences that cannot lead to feature interactions in the specification. These *false* incoherences are characterized by the fact that the tool identifies a possible incoherence between two features X and Y that are respectively in states S_m and S_n while, in fact, this situation cannot occur in the specification. This fact shows a limitation of our method.

An example of such an incoherence is the following: Alice subscribes to Voice Mail. Bob subscribes to Three Way Calling. Alice is involved in a three way conversation and at the same time, plays back the messages she received with Voice Mail. If Alice goes onhook, results are contradictory since both features react differently. However, Alice can never listen to her messages and be involved in a three way conversation at the same time.

These *false* incoherences are due to the algorithm identifying an incoherence when two features have the same triggers and different or contradictory results, while their pre-conditions show incompatible states, meaning that both features should not be triggered at the same time.

This problem is not related to the rules we developed but to the way we modeled the features: incompatible states are not identified in our model. This can be done by refining the properties used to represent states and by building contradiction pairs corresponding to incompatible states. If this is done, two features presenting incompatible states would present a contradiction in their pre-conditions, and would not be considered by the rules. This emphasizes that features must be carefully modeled and that information about contradictions between properties must be clearly stated. It also demonstrates that our method forces the designer to do a precise modeling.

6.2 More Complicated Examples

So far we have presented simple examples to facilitate the understanding of our concepts. To illustrate the power of our method, more complicated examples of incoherence identifications are presented. Examples are taken from the Mitel specification. As already mentioned, the complete report of all incoherences identified can be found in appendix A.

Rule #d3 → [cw, 3] & [cw, 1] – Alice and Bob both subscribes to Call Waiting. Alice is busy talking to Bob and Carol called Alice, which implies that Carol is held by the Call Waiting feature of Alice. If Dave calls Carol, Alice's Call Waiting specifies that Carol should send a busy signal while Carol's Call Waiting specifies that Dave should be put on hold.

Rule #t1 → [ct, 3] & [ocs, 1] – Alice subscribes to Call Transfer, holds Bob and is talking to Carol. Bob subscribes to Outgoing Call Screening. Alice wants to transfer Bob to Carol. Alice's Call Transfer intention is that Bob talks to Carol while Bob's Outgoing Call Screening is that any call to Carol is blocked.

Rule #t1 → [ct, 3] & [ics, 1] – Alice subscribes to Call Transfer, holds Bob and is talking to Carol. Carol subscribes to Incoming Call Screening. Alice wants to transfer Bob to

Carol. Alice's Call Transfer intention is that Bob talks to Carol while Carol's Incoming Call Screening is that any call from Bob is denied.

6.3 Performance Evaluation

This section presents some metrics concerning the time taken to built the Prolog representation for the Mitel features as well as for the Feature Interaction Workshop features. In addition, an overview of the algorithmic complexity of the analysis is given.

6.3.1 Human Aspects Considerations

Mitel provided us with 9 features. They were split in 20 descriptions and the modeling of these descriptions took a total of 3 days. The application of the filtering process identified 43 possible incoherences. 30 mapping rules were derived to produce test suites and no interaction was found in the LOTOS specification. Table 6.3 summarizes the results. We

Features	Descriptions	Time	Incoherences	Mapping rules	Time
9	20 (2.2 / feature)	3 days	43	30	3 days

Table 6.3: Mitel Case Study: Design Time

considered the features given in the first phase of the second Feature Interaction Workshop contest. The contest instructions [42] contain the FSM definition of 10 features. We split them into 97 descriptions (9.7/feature). The specification took a total of 15 days. 149 possible incoherences were identified by our tool. Table 6.4 summarizes the results: The

Features	Descriptions	Time	Incoherences
10	97 (9.7 / feature)	15 days	149

Table 6.4: Feature Interaction Workshop Case Study: Design Time

number of descriptions per feature for the Feature Interaction Workshop is greater than for Mitel. This is due to the fact that the Mitel features were represented at a very high level, which implies a low level of details, and thus, less description parts. The Feature Interaction Workshop features were already specified in a detailed manner (using FSM), which led to more feature descriptions, to be in accordance with the level of detail of the FSM representation.

6.3.2 Algorithmic Performance

The algorithm developed consists in taking a list of feature descriptions and applying each filtering rule on all pair-wise combinations, attempting to prove that an incoherence is present. Since we consider pair-wise combinations, the analysis of n feature descriptions results in the analysis of n^2 pairs. Each pair must be analyzed with all the rules. This results in $6n^2$ combinations to treat.

Therefore, the order of the algorithmic complexity of the analysis is n^2. The curve presented in fig. 6.2 shows the function $6n^2$. The number of combinations increases considerably along with the number of features considered. Working with pair-wise combinations always leads to this order of algorithmic complexity.

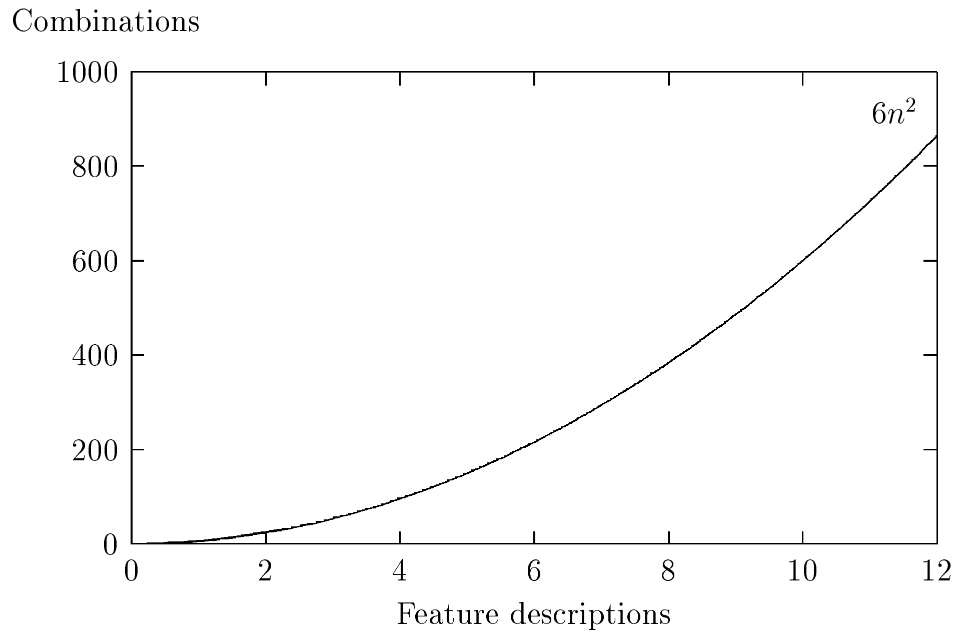

Figure 6.2: Number of Combinations over Feature Descriptions

It is not possible to reduce the order of this algorithmic complexity but Prolog interpreters are fast and the analysis of all combinations takes little time.

For a given rule r and a given combination of features $|\mathcal{D}_{X,m} \bullet \mathcal{D}_{Y,n}|$, the analysis of this combination deals with the properties and variables of both feature descriptions $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$. As mentioned, the goal is to satisfy the filtering rule. To obtain rule satisfaction, Prolog tries all possible bindings of variables until it finds a proper combination or until all combinations have been tried. The number of tries to bind variables with proper values depends on the number of users considered and on whether an incoherence is present or not.

The algorithm is built in such a way that variables of the first feature description are bound with default users, basically the first users that satisfy the constraints. Then, Prolog tries to bind variables of the second feature description with all possible combinations until an incoherence is found or everything has been tried.

Let us consider two features $\mathcal{D}_{X,m}$ and $\mathcal{D}_{Y,n}$ analyzed by a rule The analysis is done as follows:

1. Bind variables of properties of $\mathcal{D}_{X,m}$ with the first not yet tried combination of users that satisfies $\mathcal{C}_{Y,n}$ and go to step 2.

2. Value combinations remain untried ?
   – *YES* – Bind variables of properties of $\mathcal{D}_{Y,n}$ with 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 and an incoherence is identified.

7. Rule fails, no incoherence are identifiable.

The number of times that the algorithm backtracks depends on how an incoherence is identified, if ever. Given a feature description that contains n users, if an incoherence is found within the first try, all steps are visited only once. But if no incoherence exists, all combinations of users are tried and the steps are visited n^2 times until Prolog can conclude that no incoherence has been found.

Rules #d1 and #d2 can also influence the number of iterations: since they are symmetric, only one pair needs to be analyzed. Hence, the number of combinations considered by each rule is half the number of combinations considered by any of the other rules.

It is impossible to determine the exact number of iterations for a given list of feature descriptions. However, the algorithmic complexity must be computed with respect to the number of feature descriptions and the number of variables. Let us denote f the number of features and u the number of users. The algorithmic complexity of the analysis, denoted $\mathcal{C}p$, lies between a minimum and a maximum expressed by: $5f^2 \leq \mathcal{C}p \leq u^2(5f^2)$.

Considering a specification containing 4 users, the algorithmic complexity lays between $5f^2$ and $16(5f^2)$, as illustrated by the curves presented in fig. 6.3.

Thus, the algorithmic complexity of our tool is polynomial. Such a complexity is unfortunately not reducible. Prolog must try all possible combinations to be able to give a verdict. Even if the complexity is high, the execution time remains very reasonable. Executed on a Pentium II 450 Mhz, our tool gave results in a very short time considering the number of pairs analyzed:

- Mitel specification, 20 feature descriptions (400 pairs) and 4 users: 70,122,509 inferences in 84.14 seconds. 43 possible incoherences found.

- FIW00 specification, 97 feature descriptions (9409 pairs) and 4 users: 789,818,572 inferences in 1299.93 seconds. 149 possible incoherences found.

Considering that our tool needs 21 minutes and 40 seconds to analyze 9409 description pairs and find 149 possible incoherences, we can conclude that the execution time remains very reasonable in spite of the algorithmic complexity presented.

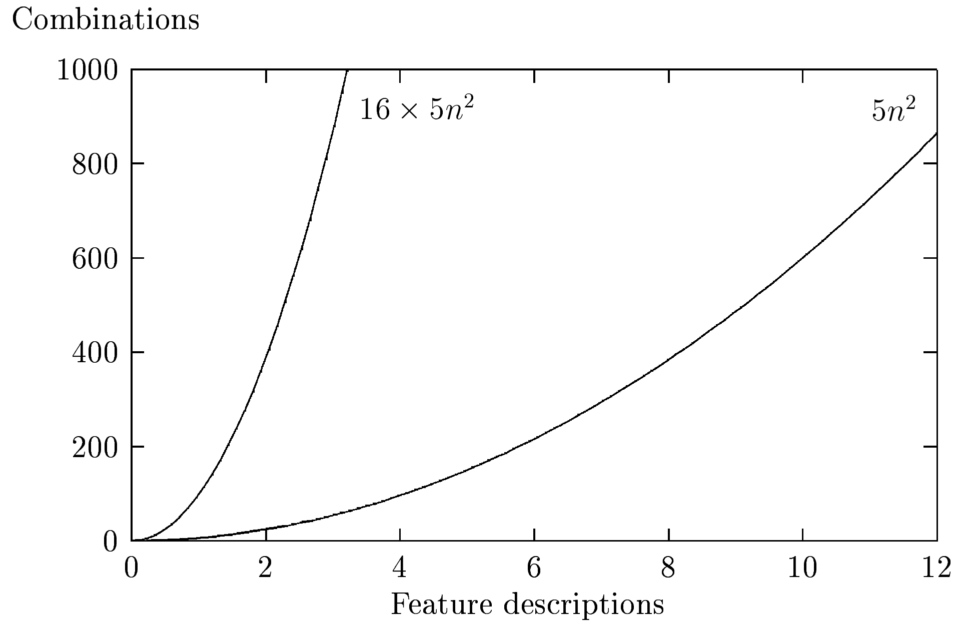

Figure 6.3: Range of Possible Complexity

6.4 Comparison

Tadashi Ohta and Tae Yoneda developed a similar method [30, 31], already mentioned in chapter 2, section 2.2.6. The most recent version of their approach [31] is illustrated in fig. 6.4. This approach starts with the specification of features using STR [44]. The syntax of STR,

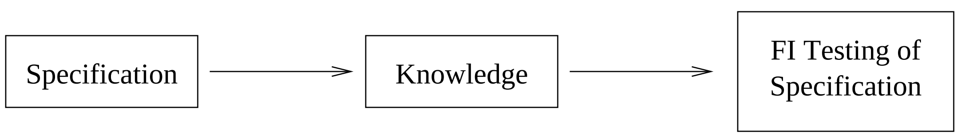

Figure 6.4: Approach Overview

composed of pre-conditions, event, and post-conditions is very similar to the one we use, which is also composed of pre-conditions, triggering-events and results. The specification is analyzed by an automatic tool that *elicits* knowledge relative to the combination of features. Such knowledge is then used to manually identify feature interactions and thus validate the specification.

The knowledge is modeled using states and transitions. A Service is represented as a set of states and rules. A system state is represented as a set of state elements called state descriptions primitives (simply called primitives).

Fig. 6.5 illustrates a case where primitives `calling(A, C)`, `idle(B)` and `m-cfv(B, C)` represent actions involved in the call forwarding and were the system goes in a state where the call from A to C is established and C rings.

A new service is added by adding new state transitions. The new transitions take precedence over the previously existing ones. Previous transitions and states to which they lead are called inhibited primitives while the new ones are called intention primitives.

Let us consider a state S_i and a rule R_e going from this state to another state S_{i+1}. Let us consider the addition of a feature implementing a new functionality. This addition,

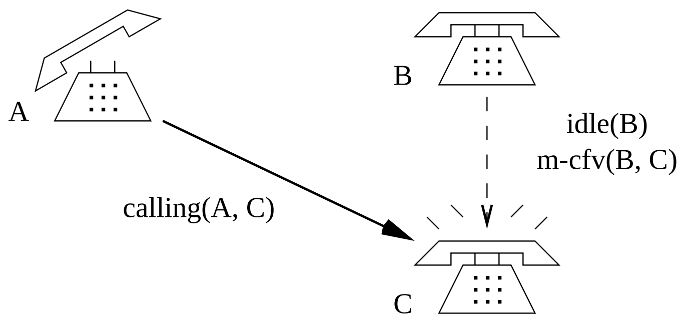

Figure 6.5: Knowledge Formal Definition: System State

illustrated by fig. 6.6, is represented as a new rule R_a going from state S_i to a new state S_j: rule R_a replaces rule R_e and thus, state P_j replaces state P_i. P_j becomes an inhibited primitive and P_j is an intention primitive. As previously mentioned, feature specifications

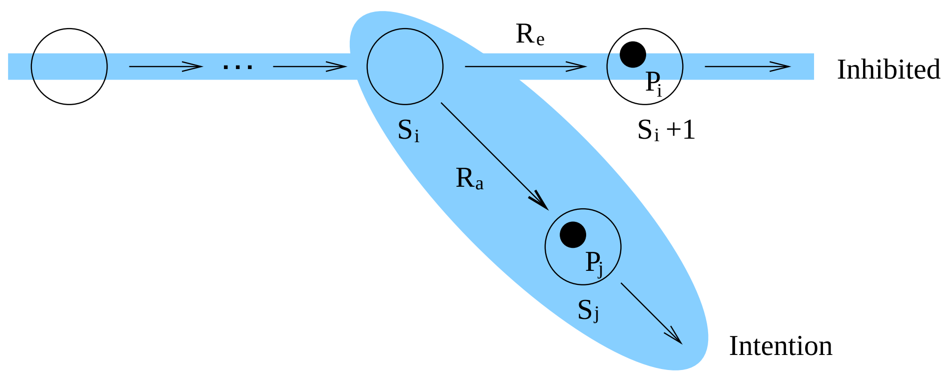

Figure 6.6: Knowledge Formal Definition: Primitives

are represented using a description language called STR (State Transition Rule) [44]. A feature is modeled using pre-conditions, events and post-conditions. This representation is very close to the one we use, based on pre-conditions, triggering events and results. However, since a feature is triggered by an event when the system is in a certain state (pre-conditions) and results in some new state (post-conditions), using STR seems to be the most appropriate to represent features.

Based on STR, T. Ohta and T. Yoneda have implemented an algorithm that *elicits* knowledge from the specification. According to our understanding, *elicitation* denotes the extraction of knowledge for the identification of inhibited and intention primitives with respect to the combinations of features. The extraction of this knowledge is automatic. Such knowledge is later manually analyzed to identify feature interactions. The interactions are characterized by specific rules. This rules are given in [31] and follow:

- *An intention primitive for one service is defined as an inhibited primitive in the other service.*

- *An inhibited primitive for one service (suppose service A) is an element of the post-pondition of a rule belonged to the other service and the rule takes precedence over the rule belonged to service A.*

- *When a rule of service A, r_a, does not take precedence over a rule of service B, r_b, r_b has not an intention primitive for service A in post-condition while r_b has the same event which is a condition for intention primitive of service A and has the same primitive in pre-condition which is a condition for intention primitive of service A.*

These rules are similar to those used in our approach. Such similarity is not surprising since features described with similar notation should lead to similar procedures. However, even if similar in the notation and rules, the approach of T. Ohta and T. Yoneda and the one we developed present substantial differences located at different stages of the process and even in the process itself. The comparison can be summarized as follows:

Design stages covered – T. Ohta and T. Yoneda method acts at the specification and testing levels while ours acts at requirements, specification and also testing levels.

Automation – T. Ohta and T. Yoneda have automatic knowledge *elicitation* but identify interactions manually (automation is considered in future work); we have automatic identification of incoherences as well as automatic test derivation, based on manual elaboration of mapping rules.

Types of FI detected – T. Ohta and T. Yoneda consider interactions caused by non-determinism and contradictions but do not seem to consider those caused by loops. We consider the three kinds.

These two approaches remain different despite their similarities. T. Ohta and T. Yoneda's method works at the specification level. Knowledge is *elicited* from the specification and feature interactions are detected in the specification. Our approach works both at requirements and specification level. It offers a filtering process that can be applied either to the requirements or to the specification, and allows the derivation of test suites that can be used to validate the specification (and even the final product).

Moreover, the filtering of incoherences is automatic and the derivation of test suites are semi-automatic, while the approach developed by T. Ohta and T. Yoneda is semi-automated in the sense that, if knowledge *elicitation* is automatic, identification of feature interactions remain manual.

6.5 In Summary

In this chapter, we illustrated the applications of our tool with examples based on the Mitel specification and the Feature Interaction Workshop. We also presented an overview of the algorithmic complexity. Results show that, even if highly complex in practice, the execution of our tool is timely reasonable. In addition, we presented a comparison with a similar approach by other authors.

Chapter 7

Conclusion & Future Work

This chapter recapitulates the thesis and lists our contributions. It also presents a number of ideas about further research and improvements of our method.

7.1 Achievements

This thesis addresses the feature interaction problem in the research area of telephony systems design. Starting with a review of this problem, we presented our definition of a feature and of a feature interaction. We enumerated and discussed a list of some known approaches with respect to their strengths and limitations. In addition to these approaches, we presented the initial approach as well as our proposed refinements.

The initial validation approach consists in deriving a Use Case Maps model from the requirements, allowing the designer to do a first validation. This model is in turn derived into a corresponding LOTOS specification. It provides an executable model and allows designers (and testers) to perform various kinds of validations (exploration or scenario based). Scenario based validation is used to perform feature interaction detection. However, this approach is based on manual validation of UCM as well as on manual derivation of test suites for feature interactions detection.

Our research mainly consists of the automation of this method. We proposed refinements that consist of a tool targeting the automatic identification of incoherences between features at the requirements stage as well as the automatic derivation of test suites. These test suites target the validation of a specification in order to check whether or not interactions corresponding to incoherences identified in requirements lead to feature interactions in the specification.

The filtering process is based on rules characterizing incoherences. Feature descriptions are formalized using information available in the requirements. Our tool performs pair-wise analysis of features and identifies pairs that satisfy one or more incoherence rules. This identification of pairs presenting incoherences allows the designer to refine the requirements and to produce a better UCM model using the knowledge available about possible interactions. Moreover, by transitivity, this also insures a better LOTOS model. Moreover, the same rules might be used for both UCM to LOTOS and test suites generation.

Our derivation of validation test suites (which follows the derivation of the LOTOS specification) can be seen as a semi-automatic process due to the fact that it requires the designer to manually provide mapping rules to go from the formal representation of features to LOTOS scenarios. Nevertheless, the elaboration of these mapping rules is fast for a designer that knows the LOTOS model.

The rules are used to produce validation test suites corresponding to the potential interacting pairs. The test suites are then applied against the LOTOS model, performing scenario based validation, to check whether the interactions corresponding to the incoherences identified exist or not in the specification. Derivation of test suites is not specific to LOTOS and mapping rules could be derived to produce scenarios in other languages.

Mitel Corp. provided us with a UCM model from which we derived a LOTOS specification. That is to say, our filtering process could not influence the requirements or the UCM model. We refined our approach and built our tool after the derivation of the LOTOS specification and the integration of 6 of the 9 features. This gave us the opportunity to identify interactions that were not suspected by Mitel's designers.

In the future, the order of these activities should be reversed in accordance with the approach presented in by figure 2.2 in chapter 2, section 2.4.

We applied our filtering process, identified incoherences and produced (via mapping rules) validation test suites for the first 6 features. Applied against the LOTOS specification, the test suites revealed that the model was free of feature interactions, meaning that all potential interactions were avoided by Mitel designers at the UCM level. However, all incoherences identified were incoherences that could have led to interactions in a system and all the scenarios derived were successfully applied against the specification, showing that no interactions were present.

In order to extend our benchmark, we applied the filtering process to the specification of the Feature Interaction Workshop 2000 (FIW00). The application of this process gave us interesting results. Contrary to the results obtained with Mitel's specification, those obtained with the FIW00 specification showed the identification of some impossible incoherences. An analysis of the results brought us to the conclusion that these results were due to a poor formalization of the features. Such results show that our method enforces a clean design and formalization of the features, which we consider as being a strength.

7.2 Contributions

This section lists our contributions to the software engineering area and to the feature interaction problem in the research area of telephony systems design.

7.2.1 Translation of a UCM Model into a LOTOS Specification

The translation of UCM into LOTOS was first proposed by D. Amyot in [21]. Chapter 3 of the thesis presents the translation of the system provided by Mitel Corp. This system is modeled as a set of UCM composed of 110 different maps distributed in seven different levels sub-maps and contains 9 feature definitions.

We provided explanations about the analysis and decisions (chapter 3, section 3.3.2) and established general translation guidelines (chapter 3, section 3.3.2) that are applicable to other systems. In addition, we have presented the approach used for integrating features to the specification.

These achievements show that the translation of such a large UCM model into a LOTOS specification is possible with reasonable efforts. The construction of the specification for the purpose of filtering is also easy and quick, as shown in chapter 6, section 6.3.1. In addition, it provides useful information that can be used by designers to refine requirements and to avoid interactions in the specification.

7.2.2 Incoherences Filtering Process

We proposed an approach that allows the automatic identification of incoherences between features at the requirements level. Chapter 4 presents the rules used for the identification of such incoherences. The tool implementing the automatic application of the rules is presented in chapter 5.

This improvement allows a faster identification of potential feature interactions with less human intervention than in the manual approaches.

7.2.3 Automatic Derivation of Validation Test Suites

Identifying incoherences at the requirements stage does not insure that these correspond to interactions in the specification.

The use of mapping rules, presented in chapter 5, section 5.3 enables the automatic derivation of test suites with respect to a given specification. These test suites are applied against the specification to insure that this one does not contain the feature interactions corresponding the the incoherences identified by the filtering.

This reduces the testing time in the sense that, once the mapping rules are manually specified, tests can be automatically derived for all incoherences identified without requiring further manual intervention. This process, which we demonstrated by using LOTOS, is not strictly dependent on the use of LOTOS; other specification languages could be used.

7.2.4 Scenario Based Validation of a Large LOTOS Specification

The manual validation, based on test scenarios derived from UCM, is presented in chapter 5, section 5.1.1. The automatic validation, based on test suites obtained from the filtering results via the use of mapping rules, is presented in chapter 5, section 5.3.

Both validations are done by applying the scenarios against the LOTOS specification using LOLA. This technique is presented in chapter 5, section 5.1.2.

These validations performed on the specification show that the scenario based validation of a large LOTOS specification is indeed possible.

7.2.5 Use of LOTOS in an Industrial Context

The translation of the UCM model resulted in a LOTOS specification that is 2410 lines long. This specification breaks down into 450 lines of Abstract Data Types (ADT) and 1960 lines of behavior.

By translating such a UCM model into a LOTOS specification and by performing feature interaction detection on this specification, we showed that the use of LOTOS in an industrial context is appropriate. The information about analysis and decisions as well the translation guidelines presented in chapter 3 show that building a LOTOS specification from a UCM model is not a complicated task.

7.2.6 Applications & Conclusive Results over two Case Studies

The filtering process is dedicated to the requirements and does not concern the validation of the specification except in the sense that it provides information that helps the designers to derive a better specification from requirements. Hence, the specification that is derived should not contain any of the interactions corresponding to the incoherences identified by the filtering process. Since the designers are aware of them, they are expected to derive the specification in such a way that interactions are avoided.

The only limitation is that the filtering process does not insure that all possible interactions are detected. This fact especially concerns multi-way incoherences since only pair-wise combinations are considered. This implies that the absence of incoherences does not insure the absence of interactions in the specification.

In spite of these considerations, the results obtained from the application of our approach over two different case studies showed that this methodology improves the design and validation in the specification process. These results are presented and commented in chapter 6. Moreover, we have proposed a representation of features and a filtering approach that is easy to use. This approach only requires knowledge of Prolog and of the language used for the specification to derive the mapping rules.

It should also be noted that in the industrial case study with Mitel corp., an interaction that was unknown to Mitel's designers was found. Since this case study was based on well known interactions, this gives hope that more interactions could be found in a set of unknown interactions.

7.3 Further Research

Many improvements are still possible on our method. This section presents those that are being considered.

7.3.1 Priorities

As presented in chapter 4, section 4.3, many incoherences can be avoided by using a priority mechanism. However, the way that priorities are currently represented by using contradiction pairs is not sufficient to model complex priority schemes.

In chapter 4, section 4.3.2, we provide an idea of how this problem could be solved.

7.3.2 Multi-way Incoherences

The filtering process only considers pair-wise combinations, and thus, only identifies two-way incoherences. Even if this allows the identification of a significant number of them, identifying three-way or even n-way incoherences is an improvement that must be considered. This requires the establishment of new incoherence rules for characterizing these kinds of incoherences.

Similarly to two-way incoherences, a simple three-way incoherence that can be identified is the direct incoherence present between three features held by the same user. A Three-way rule can identify incoherences occurring when the features do not present any contradiction in their preconditions, have the same triggering events, and have different or contradictory results.

7.3.3 Distribution over a Many Hosts Network

Since our tool combines features in a pair-wise manner, the distribution of the analysis over a many hosts network could be an improvement. Distributing the analysis of features over different hosts can significantly reduce the global analysis time since the workload would be shared.

However, due to the communication cost concerning the transmissions of feature descriptions and reports, the distributed analysis of a small group of features can possibly take more time than a centralized one. Nevertheless, the distribution of the tool would increase, in most cases, its power by reducing its execution time over most feature sets.

7.3.4 Considering other methods

The automatic analysis of incoherences is based on Prolog. However, it appears that the use of a predicate for incoherences (or contradictions) is unfortunately one of the few things that one can do to handle negation in traditional Prolog. Refining the rules using newer techniques such as linear logic [45] would possibly allow to express them in a more efficient way.

Another consideration is to link our rules to the OPI (Obligation, Permission, Interdiction) [34] model. Refining rules based on this model will, for instance, allow to make a distinction between *Obliged*, *Permitted* and *Forbidden* results of a feature.

Bibliography

[1] P. Zave. http://www.research.att.com/∼pamela/faq.html. FAQ Sheet on Feature Interaction.

[2] P. Zave. Architectural solutions to feature-interaction problems in telecommunications. In *Feature Interactions in Telecommunications and software systems VI*, pages 10–22. IOS Press, 1998.

[3] K. Kimbler L.G. Bouma (editors). *Feature Interactions in Telecommunications and software systems VI*. IOS Press, 1998.

[4] M. Calder E.H. Magill. *Feature Interactions in Telecommunications and software systems VI*. IOS Press, 2000.

[5] S.L. Pfleeger. *Software Engineering, Theory and Practice*. Prentice Hall, 1998.

[6] T. Bolognesi E. Briskma. Introduction to the ISO specification language LOTOS. In *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science, 1989.

[7] J. Ellsberger D. Hogrefe A. Sarma. *SDL Formal Object-Oriented Language for Communicating Systems*. Prentice Hall, 1997.

[8] K.J. Turner. *Using Formal Description Techniques*. J. Wiley & sons Ltd., 1993.

[9] R.J.A. Buhr D. Amyot M. Elammari D. Quesnel T. Gray S. Mankovski. Feature-interaction visualization and resolution in an agent environment. In *Feature Interactions in Telecommunications and software systems V*, pages 135–149. IOS Press, 1998.

[10] D. Marples E.H. Magill. The use of rollback to prevent incorrect operation of features in intelligent n etwork based systems. In *Feature Interactions in Telecommunications and software systems V*, pages 115–134. IOS Press, 1998.

[11] P. Combes S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In *Feature Interaction In Telecommunications Systems*, pages 120–135. IOS press, 1994.

[12] M. Faci L. Logrippo. An algebraic framework for the feature interaction problem. In *Proc. of the 3rd AMAST Workshop on Real-Time Systems*, pages 280–294, 1996.

[13] E. Kit. *Software Testing in the Real World, Improving the Process*. Addison Wesley, 1996.

[14] G. Utas. A pattern language of feature interaction. In *Feature Interactions in Telecommunications and software systems V*, pages 98–114. IOS Press, 1998.

[15] R.J.A. Buhr R.S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.

[16] R.J.A. Buhr. Use Case Maps as architectural entities for complex systems. In *IEEE Transactions on Software Engineering (TSE)*, pages 1131–1155, 1998.

[17] D. Amyot. http://www.usecasemaps.org/. Use Case Maps Web Page and UCM User Group.

[18] D. Amyot R.J.A Buhr T. Gray. Use Case Maps for the capture and validation of distributed systems requirements. In *Fourth International Symposium on Requirements Engineering (RE'99)*, 1999.

[19] D. Amyot L. Logrippo. Use Case Maps and LOTOS for the prototyping and validation of GPRS Group-Call. In *Computer Communications 23(8)*, 1999.

[20] D. Amyot L. Charfi N. Gorse T. Gray L. Logrippo. J. Sincennes B. Stepien T. Ware. Feature description and feature interaction analysis with Use Case Maps and LOTOS. In *Feature Interactions in Telecommunications and software systems VI*, pages 274–289. IOS Press, 2000.

[21] D. Amyot. Formalization of timethreads using LOTOS. Master's thesis, University of Ottawa, Ottawa, 1994.

[22] J. Kamoun L. Logrippo. Goal oriented feature interaction detection in the intelligent network model. In *Feature Interactions in Telecommunications and software systems V*, pages 172–186. IOS Press, 1998.

[23] A. Valmari. The state explosion problem. lectures on petri nets i: Basic models. In *Lecture Notes in Computer Science 1491*, pages 429–528. Springer-Verlag, 1998.

[24] J.E. Hopcroft J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[25] J. L. Peterson. An introduction to petri nets. In *Proc. of the National Electronics Conference, Chicago, Illinois*, pages 144–148. National Engineering Consortium, Inc., 1978.

[26] M. Nakamura Y. Katuda T. Kikuno. Feature interaction detection using permutation symmetry. In *Feature Interactions in Telecommunications and software systems V*, pages 187–201. IOS Press, 1998.

[27] Z. Manna A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

[28] A.P. Felty K.S. Namjoshi. Feature specification and automatic conflict detection. In *Feature Interactions in Telecommunications and software systems VI*, pages 179–192. IOS Press, 2000.

[29] R.H. Hardin Z. Har'el R.P. Kurshan. Cospan. In *Conference on computer aided verification*, 1996.

[30] T. Yoneda T. Ohta. A formal approach for definitions and detection of feature interactions. In *Feature Interactions in Telecommunications and software systems V*, pages 202–216. IOS Press, 1998.

[31] T. Yoneda T. Ohta. Automatic elicitation of knowledge for detecting feature interactions in telecommunications services. In *IEICE Transactions on Information and Systems*, pages 640–647. IEICE, Japan, 2000.

[32] E. Rudolph P. Graubmann J. Grabowski. Tutorial on message sequence charts. http://http://www.sdl-forum.org/MSC/index.htm.

[33] J. Grabowski A. Wiles C. Willcock D. Hogrefe. On the design of the new testing language ttcn-3. In *Testing of communicating systems tools and techniques*, pages 161–176. KAP, 2000.

[34] M. Barbuceanu T. Gray S. Mankowski. How to make your agents fulfil their obligations. In *Proceedings of PAAM-98*, 1998.

[35] S. Pavón D. Larrabeiti G. Rabay. LOtos LAboratory user manual (version 3r6). Universidad Politécnica de Madrid, 1995.

[36] Groupe d'Intelligence Artificielle, université de Marseilles, France. *Prolog: Manuel de Reference et d'Utilisation*, 1975.

[37] W.F. Clocksin C.S. Mellish. *Programming in Prolog.* Springer, Fourth Edition.

[38] A. Miga. Application of Use Case Maps to system design with tool support. Master's thesis, University of Carleton, Ottawa, 1998.

[39] C.A. Vissers G. Scollo M. van Sinderen E. Brinksma. Specification styles in distributed systems design and verification basic models. In *Lecture Notes in Computer Science 1491*, pages 179–206. Theoritical Computer Science '89, 1989.

[40] P.H. Winston. *Artificial Intelligence, Third Edition, chapter 13.* Addison Wesley, 1992.

[41] J. Wielemaker. http://swi.psy.uva.nl/projects/SWI-prolog/. SWI-Prolog.

[42] M. Kolberg E.H. Magill D. Marples S. Reiff. Second feature interaction contest, instructions. In *Feature Interactions in Telecommunications and software systems VI*, pages 293–325. IOS Press, 2000.

[43] E.H. Magill. http://www.comms.eee.strath.ac.uk/∼fi/fiw00/. Feature Interaction Workshop 2000.

[44] T. Yoneda T. Ohta. Declarative language str (state transition rule). In *FIRE works Workshop on Language Construcs for Describing Features*, pages 39–54. S. Gilmore & M. Ryan, 2000.

[45] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*. Springer Verlag LNCS 711, 1993.

Appendix A

Filtering Results for Mitel

```
* Rule #d1 -> [cfb, 1] & [ics, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [ics, 1]
            subs(bob, ics)
            concerns(bob, ics)
            ics_list(alice)

     + Same triggering events
         call(alice, bob)

     + Different results

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

        - Resulting events of [ics, 1]
            deny_call(bob, alice)
            call_denied(bob, alice)
```

```
* Rule #d1 -> [cfb, 1] & [cw, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [cw, 1]
            subs(bob, cw)
            concerns(bob, cw)
            busy(bob)
            talk(bob, carole)

     + Same triggering events
         call(alice, bob)

     + Different results

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

        - Resulting events of [cw, 1]
            hold(bob, alice)
            cw_notify(bob)
```

```
* Rule #d1 -> [cfb, 1] & [cw, 4]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [cw, 4]
            subs(bob, cw)
            concerns(bob, cw)
            busy(bob)
            talk(bob, carole)
            hold(bob, dave)

     + Same triggering events
         call(alice, bob)

     + Different results

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

        - Resulting events of [cw, 4]
            busy_ind(bob, alice)
```

```
* Rule #d1 -> [cfb, 1] & [arc, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [arc, 1]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)

     + Same triggering events
         call(alice, bob)

     + Different results

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

        - Resulting events of [arc, 1]
            store(bob, alice)
            arc_notify(alice)
```

```
* Rule #d1 -> [cfa, 1] & [ics, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

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

        - Pre-conditions of [ics, 1]
            subs(bob, ics)
            concerns(bob, ics)
            ics_list(alice)

     + Same triggering events
         call(alice, bob)

     + Different results

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Resulting events of [ics, 1]
            deny_call(bob, alice)
            call_denied(bob, alice)
```

```
* Rule #d1 -> [cfa, 1] & [cw, 4]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

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

        - Pre-conditions of [cw, 4]
            subs(bob, cw)
            concerns(bob, cw)
            busy(bob)
            talk(bob, carole)
            hold(bob, dave)

     + Same triggering events
         call(alice, bob)

     + Different results

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Resulting events of [cw, 4]
            busy_ind(bob, alice)
```

```
* Rule #d1 -> [cfa, 1] & [cw, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

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

        - Pre-conditions of [cw, 1]
            subs(bob, cw)
            concerns(bob, cw)
            busy(bob)
            talk(bob, carole)

     + Same triggering events
         call(alice, bob)

     + Different results

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Resulting events of [cw, 1]
            hold(bob, alice)
            cw_notify(bob)
```

```
* Rule #d1 -> [cfa, 1] & [arc, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

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

        - Pre-conditions of [arc, 1]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)

     + Same triggering events
         call(alice, bob)

     + Different results

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Resulting events of [arc, 1]
            store(bob, alice)
            arc_notify(alice)
```

```
* Rule #d1 -> [ocs, 1] & [ct, 2]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [ocs, 1]
            subs(alice, ocs)
            concerns(alice, ocs)
            ocs_list(bob)

        - Pre-conditions of [ct, 2]
            subs(alice, ct)
            concerns(alice, ct)
            busy(alice)
            hold(alice, carole)
            dialtone(alice)

     + Same triggering events
         call(alice, bob)

     + Different results

        - Resulting events of [ocs, 1]
            block_call(alice, bob)
            call_blocked(alice, alice)

        - Resulting events of [ct, 2]
            talk(alice, bob)
```

```
* Rule #d1 -> [ct, 1] & [cw, 2]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [ct, 1]
            subs(alice, ct)
            concerns(alice, ct)
            busy(alice)
            talk(alice, bob)

        - Pre-conditions of [cw, 2]
            subs(alice, cw)
            concerns(alice, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

     + Same triggering events
         flash(alice)

     + Different results

        - Resulting events of [ct, 1]
            hold(alice, bob)
            dialtone(alice)

        - Resulting events of [cw, 2]
            hold(alice, bob)
            talk(alice, carole)
```

```
* Rule #d1 -> [ics, 1] & [arc, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [ics, 1]
            subs(bob, ics)
            concerns(bob, ics)
            ics_list(alice)

        - Pre-conditions of [arc, 1]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)

     + Same triggering events
         call(alice, bob)

     + Different results

        - Resulting events of [ics, 1]
            deny_call(bob, alice)
            call_denied(bob, alice)

        - Resulting events of [arc, 1]
            store(bob, alice)
            arc_notify(alice)
```

```
* Rule #d1 -> [cw, 1] & [arc, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [cw, 1]
            subs(alice, cw)
            concerns(alice, cw)
            busy(alice)
            talk(alice, bob)

        - Pre-conditions of [arc, 1]
            subs(alice, arc)
            concerns(alice, arc)
            busy(alice)

     + Same triggering events
         call(carole, alice)

     + Different results

        - Resulting events of [cw, 1]
            hold(alice, carole)
            cw_notify(alice)

        - Resulting events of [arc, 1]
            store(alice, carole)
            arc_notify(carole)
```

```
* Rule #d1 -> [cw, 4] & [arc, 1]
  Same user subscribed to two features having the
  same triggering events and different results

     + Features pre-conditions

        - Pre-conditions of [cw, 4]
            subs(alice, cw)
            concerns(alice, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

        - Pre-conditions of [arc, 1]
            subs(alice, arc)
            concerns(alice, arc)
            busy(alice)

     + Same triggering events
         call(dave, alice)

     + Different results

        - Resulting events of [cw, 4]
            busy_ind(alice, dave)

        - Resulting events of [arc, 1]
            store(alice, dave)
            arc_notify(dave)
```

```
* Rule #d2 -> [cfb, 1] & [cfa, 1]
  Same user subscribed 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(carole)
            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, carole)
            call(alice, carole)
            ring(alice, carole)

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

```
* Rule #d2 -> [ics, 1] & [cw, 1]
  Same user subscribed to two features having the
  same triggering events and contradictory results

     + Features pre-conditions

        - Pre-conditions of [ics, 1]
            subs(bob, ics)
            concerns(bob, ics)
            ics_list(alice)

        - Pre-conditions of [cw, 1]
            subs(bob, cw)
            concerns(bob, cw)
            busy(bob)
            talk(bob, carole)

     + Same triggering events
         call(alice, bob)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(bob, alice)
            call_denied(bob, alice)

        - Resulting events of [cw, 1]
            hold(bob, alice)
            cw_notify(bob)
```

```
* Rule #d2 -> [ics, 1] & [cw, 4]
  Same user subscribed to two features having the
  same triggering events and contradictory results

     + Features pre-conditions

        - Pre-conditions of [ics, 1]
            subs(bob, ics)
            concerns(bob, ics)
            ics_list(alice)

        - Pre-conditions of [cw, 4]
            subs(bob, cw)
            concerns(bob, cw)
            busy(bob)
            talk(bob, carole)
            hold(bob, dave)

     + Same triggering events
         call(alice, bob)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(bob, alice)
            call_denied(bob, alice)

        - Resulting events of [cw, 4]
            busy_ind(bob, alice)
```

```
* Rule #d3 -> [cw, 3] & [cfb, 1]
  Different users subscribed to two features where
  feature 1 concerns subscriber of feature 2, and
  the features have the same triggering events and
  different results

     + Features pre-conditions

        - Pre-conditions of [cw, 3]
            subs(alice, cw)
            concerns(carole, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

        - Pre-conditions of [cfb, 1]
            subs(carole, cfb)
            concerns(carole, cfb)
            cfb(alice)
            busy(carole)

     + Same triggering events
         call(dave, carole)

     + Different results

        - Resulting events of [cw, 3]
            busy_ind(carole, dave)

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

```
* Rule #d3 -> [cw, 3] & [cw, 1]
  Different users subscribed to two features where
  feature 1 concerns subscriber of feature 2, and
  the features have the same triggering events and
  different results

     + Features pre-conditions

        - Pre-conditions of [cw, 3]
            subs(alice, cw)
            concerns(carole, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

        - Pre-conditions of [cw, 1]
            subs(carole, cw)
            concerns(carole, cw)
            busy(carole)
            talk(carole, alice)

     + Same triggering events
         call(dave, carole)

     + Different results

        - Resulting events of [cw, 3]
            busy_ind(carole, dave)

        - Resulting events of [cw, 1]
            hold(carole, dave)
            cw_notify(carole)
```

```
* Rule #d3 -> [cw, 3] & [cfa, 1]
  Different users subscribed to two features where
  feature 1 concerns subscriber of feature 2, and
  the features have the same triggering events and
  different results

     + Features pre-conditions

        - Pre-conditions of [cw, 3]
            subs(alice, cw)
            concerns(carole, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

        - Pre-conditions of [cfa, 1]
            subs(carole, cfa)
            concerns(carole, cfa)
            cfa(alice)

     + Same triggering events
         call(dave, carole)

     + Different results

        - Resulting events of [cw, 3]
            busy_ind(carole, dave)

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

```
* Rule #d3 -> [cw, 3] & [arc, 1]
  Different users subscribed to two features where
  feature 1 concerns subscriber of feature 2, and
  the features have the same triggering events and
  different results

     + Features pre-conditions

        - Pre-conditions of [cw, 3]
            subs(alice, cw)
            concerns(carole, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

        - Pre-conditions of [arc, 1]
            subs(carole, arc)
            concerns(carole, arc)
            busy(carole)

     + Same triggering events
         call(dave, carole)

     + Different results

        - Resulting events of [cw, 3]
            busy_ind(carole, dave)

        - Resulting events of [arc, 1]
            store(carole, dave)
            arc_notify(dave)
```

```
* Rule #d4 -> [cw, 3] & [ics, 1]
  Different users subscribed to two features where
  feature 1 concerns subscriber of feature 2, and
  the features have the same triggering events and
  contradictory results

     + Features pre-conditions

        - Pre-conditions of [cw, 3]
            subs(alice, cw)
            concerns(carole, cw)
            busy(alice)
            talk(alice, bob)
            hold(alice, carole)

        - Pre-conditions of [ics, 1]
            subs(carole, ics)
            concerns(carole, ics)
            ics_list(dave)

     + Same triggering events
         call(dave, carole)

     + Contradictory results

        - Resulting events of [cw, 3]
            busy_ind(carole, dave)

        - Resulting events of [ics, 1]
            deny_call(carole, dave)
            call_denied(carole, dave)
```

```
* Rule #t1 -> [cfb, 1] & [cfb, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [cfb, 1]
            subs(carole, cfb)
            concerns(carole, cfb)
            cfb(bob)
            busy(carole)

     + Transitivity between features

        - Triggering events of [cfb, 1]
            call(alice, bob)

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

        - Including triggering events of [cfb, 1]
           call(alice, carole)

     + Contradictory results

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

        - Resulting events of [cfb, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)
```

```
* Rule #t1 -> [cfb, 1] & [cfa, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

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

     + Transitivity between features

        - Triggering events of [cfb, 1]
            call(alice, bob)

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

        - Including triggering events of [cfa, 1]
           call(alice, carole)

     + Contradictory results

        - Resulting events of [cfa, 1]
            redirected(alice, bob)
            call(alice, bob)
            ring(alice, bob)

        - Resulting events of [cfb, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)
```

```
* Rule #t1 -> [cfb, 1] & [ocs, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [ocs, 1]
            subs(alice, ocs)
            concerns(alice, ocs)
            ocs_list(carole)

     + Transitivity between features

        - Triggering events of [cfb, 1]
            call(alice, bob)

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

        - Including triggering events of [ocs, 1]
           call(alice, carole)

     + Contradictory results

        - Resulting events of [ocs, 1]
            block_call(alice, carole)
            call_blocked(alice, alice)

        - Resulting events of [cfb, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)
```

```
* Rule #t1 -> [cfb, 1] & [ics, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [ics, 1]
            subs(carole, ics)
            concerns(carole, ics)
            ics_list(alice)

     + Transitivity between features

        - Triggering events of [cfb, 1]
            call(alice, bob)

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

        - Including triggering events of [ics, 1]
           call(alice, carole)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(carole, alice)
            call_denied(carole, alice)

        - Resulting events of [cfb, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)
```

```
* Rule #t1 -> [cfa, 1] & [cfa, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

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

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

     + Transitivity between features

        - Triggering events of [cfa, 1]
            call(alice, bob)

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Including triggering events of [cfa, 1]
           call(alice, carole)

     + Contradictory results

        - Resulting events of [cfa, 1]
            redirected(alice, bob)
            call(alice, bob)
            ring(alice, bob)

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

```
* Rule #t1 -> [cfa, 1] & [cfb, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

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

        - Pre-conditions of [cfb, 1]
            subs(carole, cfb)
            concerns(carole, cfb)
            cfb(bob)
            busy(carole)

     + Transitivity between features

        - Triggering events of [cfa, 1]
            call(alice, bob)

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Including triggering events of [cfb, 1]
           call(alice, carole)

     + Contradictory results

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

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

```
* Rule #t1 -> [cfa, 1] & [ocs, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

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

        - Pre-conditions of [ocs, 1]
            subs(alice, ocs)
            concerns(alice, ocs)
            ocs_list(carole)

     + Transitivity between features

        - Triggering events of [cfa, 1]
            call(alice, bob)

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Including triggering events of [ocs, 1]
           call(alice, carole)

     + Contradictory results

        - Resulting events of [ocs, 1]
            block_call(alice, carole)
            call_blocked(alice, alice)

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

```
* Rule #t1 -> [cfa, 1] & [ics, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

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

        - Pre-conditions of [ics, 1]
            subs(carole, ics)
            concerns(carole, ics)
            ics_list(alice)

     + Transitivity between features

        - Triggering events of [cfa, 1]
            call(alice, bob)

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Including triggering events of [ics, 1]
           call(alice, carole)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(carole, alice)
            call_denied(carole, alice)

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

```
* Rule #t1 -> [ct, 3] & [cfa, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [ct, 3]
            subs(alice, ct)
            concerns(alice, ct)
            busy(alice)
            hold(alice, bob)
            talk(alice, carole)

        - Pre-conditions of [cfa, 1]
            subs(carole, cfa)
            concerns(carole, cfa)
            cfa(alice)

     + Transitivity between features

        - Triggering events of [ct, 3]
            transfer(alice)

        - Resulting events of [ct, 3]
            call(bob, carole)

        - Including triggering events of [cfa, 1]
           call(bob, carole)

     + Contradictory results

        - Resulting events of [cfa, 1]
            redirected(bob, alice)
            call(bob, alice)
            ring(bob, alice)

        - Resulting events of [ct, 3]
            call(bob, carole)
```

```
* Rule #t1 -> [ct, 3] & [cfb, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [ct, 3]
            subs(alice, ct)
            concerns(alice, ct)
            busy(alice)
            hold(alice, bob)
            talk(alice, carole)

        - Pre-conditions of [cfb, 1]
            subs(carole, cfb)
            concerns(carole, cfb)
            cfb(alice)
            busy(carole)

     + Transitivity between features

        - Triggering events of [ct, 3]
            transfer(alice)

        - Resulting events of [ct, 3]
            call(bob, carole)

        - Including triggering events of [cfb, 1]
           call(bob, carole)

     + Contradictory results

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

        - Resulting events of [ct, 3]
            call(bob, carole)
```

```
* Rule #t1 -> [ct, 3] & [ocs, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [ct, 3]
            subs(alice, ct)
            concerns(alice, ct)
            busy(alice)
            hold(alice, bob)
            talk(alice, carole)

        - Pre-conditions of [ocs, 1]
            subs(bob, ocs)
            concerns(bob, ocs)
            ocs_list(carole)

     + Transitivity between features

        - Triggering events of [ct, 3]
            transfer(alice)

        - Resulting events of [ct, 3]
            call(bob, carole)

        - Including triggering events of [ocs, 1]
           call(bob, carole)

     + Contradictory results

        - Resulting events of [ocs, 1]
            block_call(bob, carole)
            call_blocked(bob, bob)

        - Resulting events of [ct, 3]
            call(bob, carole)
```

```
* Rule #t1 -> [ct, 3] & [ics, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [ct, 3]
            subs(alice, ct)
            concerns(alice, ct)
            busy(alice)
            hold(alice, bob)
            talk(alice, carole)

        - Pre-conditions of [ics, 1]
            subs(carole, ics)
            concerns(carole, ics)
            ics_list(bob)

     + Transitivity between features

        - Triggering events of [ct, 3]
            transfer(alice)

        - Resulting events of [ct, 3]
            call(bob, carole)

        - Including triggering events of [ics, 1]
           call(bob, carole)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(carole, bob)
            call_denied(carole, bob)

        - Resulting events of [ct, 3]
            call(bob, carole)
```

```
* Rule #t1 -> [cp, 1] & [cfa, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cp, 1]
            subs(bob, cp)
            concerns(bob, cp)
            call(alice, carole)
            ring(alice, carole)
            idle(bob)

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

     + Transitivity between features

        - Triggering events of [cp, 1]
            pickup(bob)

        - Resulting events of [cp, 1]
            picked_up(alice, bob)
            call(alice, bob)

        - Including triggering events of [cfa, 1]
           call(alice, bob)

     + Contradictory results

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Resulting events of [cp, 1]
            picked_up(alice, bob)
            call(alice, bob)
```

```
* Rule #t1 -> [cp, 1] & [ocs, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cp, 1]
            subs(bob, cp)
            concerns(bob, cp)
            call(alice, carole)
            ring(alice, carole)
            idle(bob)

        - Pre-conditions of [ocs, 1]
            subs(alice, ocs)
            concerns(alice, ocs)
            ocs_list(bob)

     + Transitivity between features

        - Triggering events of [cp, 1]
            pickup(bob)

        - Resulting events of [cp, 1]
            picked_up(alice, bob)
            call(alice, bob)

        - Including triggering events of [ocs, 1]
           call(alice, bob)

     + Contradictory results

        - Resulting events of [ocs, 1]
            block_call(alice, bob)
            call_blocked(alice, alice)

        - Resulting events of [cp, 1]
            picked_up(alice, bob)
            call(alice, bob)
```

```
* Rule #t1 -> [cp, 1] & [ics, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [cp, 1]
            subs(bob, cp)
            concerns(bob, cp)
            call(alice, carole)
            ring(alice, carole)
            idle(bob)

        - Pre-conditions of [ics, 1]
            subs(bob, ics)
            concerns(bob, ics)
            ics_list(alice)

     + Transitivity between features

        - Triggering events of [cp, 1]
            pickup(bob)

        - Resulting events of [cp, 1]
            picked_up(alice, bob)
            call(alice, bob)

        - Including triggering events of [ics, 1]
           call(alice, bob)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(bob, alice)
            call_denied(bob, alice)

        - Resulting events of [cp, 1]
            picked_up(alice, bob)
            call(alice, bob)
```

```
* Rule #t1 -> [arc, 3] & [cfb, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [arc, 3]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)
            stored(alice)
            arc_ring(bob)

        - Pre-conditions of [cfb, 1]
            subs(alice, cfb)
            concerns(alice, cfb)
            cfb(carole)
            busy(alice)

     + Transitivity between features

        - Triggering events of [arc, 3]
            offhook(bob)

        - Resulting events of [arc, 3]
            call(bob, alice)

        - Including triggering events of [cfb, 1]
           call(bob, alice)

     + Contradictory results

        - Resulting events of [cfb, 1]
            redirected(bob, carole)
            call(bob, carole)
            ring(bob, carole)

        - Resulting events of [arc, 3]
            call(bob, alice)
```

```
* Rule #t1 -> [arc, 3] & [cfa, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [arc, 3]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)
            stored(alice)
            arc_ring(bob)

        - Pre-conditions of [cfa, 1]
            subs(alice, cfa)
            concerns(alice, cfa)
            cfa(carole)

     + Transitivity between features

        - Triggering events of [arc, 3]
            offhook(bob)

        - Resulting events of [arc, 3]
            call(bob, alice)

        - Including triggering events of [cfa, 1]
           call(bob, alice)

     + Contradictory results

        - Resulting events of [cfa, 1]
            redirected(bob, carole)
            call(bob, carole)
            ring(bob, carole)

        - Resulting events of [arc, 3]
            call(bob, alice)
```

```
* Rule #t1 -> [arc, 3] & [ocs, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [arc, 3]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)
            stored(alice)
            arc_ring(bob)

        - Pre-conditions of [ocs, 1]
            subs(bob, ocs)
            concerns(bob, ocs)
            ocs_list(alice)

     + Transitivity between features

        - Triggering events of [arc, 3]
            offhook(bob)

        - Resulting events of [arc, 3]
            call(bob, alice)

        - Including triggering events of [ocs, 1]
           call(bob, alice)

     + Contradictory results

        - Resulting events of [ocs, 1]
            block_call(bob, alice)
            call_blocked(bob, bob)

        - Resulting events of [arc, 3]
            call(bob, alice)
```

```
* Rule #t1 -> [arc, 3] & [ics, 1]
  The user(s) subscribed to different features and the
  results of F1 include the triggering events of F2
  but their results are contradictory

     + Different pre-conditions

        - Pre-conditions of [arc, 3]
            subs(bob, arc)
            concerns(bob, arc)
            busy(bob)
            stored(alice)
            arc_ring(bob)

        - Pre-conditions of [ics, 1]
            subs(alice, ics)
            concerns(alice, ics)
            ics_list(bob)

     + Transitivity between features

        - Triggering events of [arc, 3]
            offhook(bob)

        - Resulting events of [arc, 3]
            call(bob, alice)

        - Including triggering events of [ics, 1]
           call(bob, alice)

     + Contradictory results

        - Resulting events of [ics, 1]
            deny_call(alice, bob)
            call_denied(alice, bob)

        - Resulting events of [arc, 3]
            call(bob, alice)
```

```
* Rule #t2 -> [cfb, 1] & [cfb, 1]
  The users subscribed to different features for which
  the resulting events of F1 include triggering events
  of F2 and vice-versa, which leads to a loop.

     + Different or common pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

        - Pre-conditions of [cfb, 1]
            subs(carole, cfb)
            concerns(carole, cfb)
            cfb(bob)
            busy(carole)

     + Transitivity between features

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

        - Including triggering events of [cfb, 1]
           call(alice, carole)

     + Reverse transitivity, leading to a loop

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

        - Including triggering events of [cfb, 1]
           call(alice, bob)



* Rule #t2 -> [cfa, 1] & [cfa, 1]
  The users subscribed to different features for which
  the resulting events of F1 include triggering events
  of F2 and vice-versa, which leads to a loop.

     + Different or common pre-conditions

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

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

     + Transitivity between features

        - Resulting events of [cfa, 1]
            redirected(alice, carole)
            call(alice, carole)
            ring(alice, carole)

        - Including triggering events of [cfa, 1]
           call(alice, carole)

     + Reverse transitivity, leading to a loop

        - Resulting events of [cfa, 1]
            redirected(alice, bob)
            call(alice, bob)
            ring(alice, bob)

        - Including triggering events of [cfa, 1]
           call(alice, bob)
```

```
* Rule #t2 -> [cfb, 1] & [cfa, 1]
  The users subscribed to different features for which
  the resulting events of F1 include triggering events
  of F2 and vice-versa, which leads to a loop.

     + Different or common pre-conditions

        - Pre-conditions of [cfb, 1]
            subs(bob, cfb)
            concerns(bob, cfb)
            cfb(carole)
            busy(bob)

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

     + Transitivity between features

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

        - Including triggering events of [cfa, 1]
           call(alice, carole)

     + Reverse transitivity, leading to a loop

        - Resulting events of [cfa, 1]
            redirected(alice, bob)
            call(alice, bob)
            ring(alice, bob)

        - Including triggering events of [cfb, 1]
           call(alice, bob)
```

Appendix B

Prolog Implementation

```
%--------------------------------------------------------------------------
% Module:    fi-lookup-rules-v15
% Version:   15
% Modified:  Wed Sep 20 10:24:08 EST 2000
% Author:    Nicolas Gorse <ngorse@site.uottawa.ca>
%
%
%  Usage: fi-lookup <command>
%      * where command must be one of the following:
%      -filtering <features> [options]
%      --> find possible incoherences
%          * where options include:
%          -g <file> --- feature sets to combine
%          -k <file> --- list of already known incoherences
%          -i <file> --- file to which reports will be written
%
%      -testiso <features> <mapping-rules> [options]
%      --> produces test suites for features in isolation
%          * where options include:
%          -t <file> --- file to which test suites will be written
%
%      -testfi <mapping-rules> <known incoherences> [options]
%      --> produces test suites for given incoherences
%          * where options include:
%          -t <file> --- file to which test suites will be written
%
%      -filttest <features> <mapping-rules> [options]
%      --> finds possible incoherences & produce test suites
%          * where options include:
%          -g <file> --- feature sets to combine
%          -k <file> --- list of already known incoherences
%          -i <file> --- file to which reports will be written
%          -t <file> --- file to which test suites will be written
```

```
%
% Comments: This version is the last stable one. It contains a small
%           interface allowing interactivity with the user using the
%           command line arguments allowing also to call it from batch
%           scripts.
%----------------------------------------------------------------------------


                    %************************************%


%----------* Basic predicate definitions *--------------------------------


% Rules predicates, basicaly used to speed up the program,
% each predicate correspond to one of the six incoherence rules.
%
rule(d1).
rule(d2).
rule(d3).
rule(d4).
rule(t1).
rule(t2).


% Predicate used to convert a list of elements to a correponding
% string. The elements are concatenated from head to tail.
%
list_to_string([], String, String).

list_to_string([Elem|List], InString, OutString):-
string_concat(InString, Elem, TmpString),
list_to_string(List, TmpString, OutString), !.


% Predicate used to write time informations about the starting time
% and stopping time of an analysis. Useful to collect statistics.
%
write_time(start):-
get_time(T), convert_time(T, Yr, Mth, Day, Hr, Min, Sec, MSec),
write('\t--- start time: '),
write(Day), write('/'), write(Mth), write('/'), write(Yr),
write(' at '), write(Hr), write(':'), write(Min), write(':'),
write(Sec), write('.'), write(MSec).
```

```
write_time(stop):-
get_time(T), convert_time(T, Yr, Mth, Day, Hr, Min, Sec, MSec),
write('\t--- stop time : '),
write(Day), write('/'), write(Mth), write('/'), write(Yr),
write(' at '), write(Hr), write(':'), write(Min), write(':'),
write(Sec), write('.'), write(MSec).


% Predicate used to output a list to the screen. It prints the
% specified "tabulation" characters followed by an element.
% The elements are printed one by one from head to tail.
%
writeSet(_, []).

writeSet(TAB, [Elem|SubSet]):-
write(TAB), write(Elem), nl,
writeSet(TAB, SubSet), !.


% Predicate used to build the sets of features to be compared.
% It takes the definitions of all features and builds sets that
% contain the complete list of features.
%
build_feature_groups:-
feature(X, _, _, _),
not(lfeature(X)),
assert(lfeature(X)),
assert(rfeature(X)),
build_feature_groups, !.

build_feature_groups:-
feature(X, _, _, _), lfeature(X).

%----------* Basic predicate definitions *----------------------------------


                    %*************************************%


%----------* Arguments predicate definitions *-----------------------------

% Predicate managing the case of empty arguments command line
%
scan_args([]):-
print_usage.
```

```
% Predicate managing the argument concerning the incoherences
% analysis without test suites generation
%
scan_args(['-filtering', Features|Options]):-
exists_file(Features),
consult(Features),
scan_fi_options(Options), (
  fi_output_file(Output),
  open(Output, write, Stream),
  set_output(Stream)
           ;
          not(fi_output_file(_)),
  stdout(Stream), set_output(Stream)
        ), (
  groups_loaded(yes), time(fi_search), halt
           ;
  not(groups_loaded(yes)),
  build_feature_groups, time(fi_search), halt
        )
 ;
not(exists_file(Features)),
write('\t--- Invalid file: '),
write(Features), nl, halt.


% Predicate managing the argument concerning the test suites
% generation for features in isolation
%
scan_args(['-testiso', Features, Mapping|Options]):-
exists_file(Features),
consult(Features), (
  exists_file(Mapping),
  consult(Mapping),
  scan_iso_options(Options), (
    iso_output_file(Output),
    open(Output, write, Stream),
    set_output(Stream)
             ;
            not(iso_output_file(_)),
    stdout(Stream), set_output(Stream)
          ), iso_gen_test, halt
   ;
  not(exists_file(Mapping)),
```

```
  write('\t--- Invalid file: '),
  write(Mapping), nl, halt
        )
 ;
not(exists_file(Features)),
write('\t--- Invalid file: '),
write(Features), nl, halt.


% Predicate managing the argument concerning the test suites
% generation without incoherence analysis
%
scan_args(['-testfi', Mapping, Known|Options]):-
exists_file(Mapping),
consult(Mapping), (
  exists_file(Known),
  consult(Known),
  scan_ts_options(Options), (
    ts_output_file(Output),
    open(Output, write, Stream),
    set_output(Stream)
            ;
            not(ts_output_file(_)),
    stdout(Stream), set_output(Stream)
         ), fi_gen_test, halt
   ;
  not(exists_file(Known)),
  write('\t--- Invalid file: '),
  write(Known), nl, halt
        )
 ;
not(exists_file(Mapping)),
write('\t--- Invalid file: '),
write(Mapping), nl, halt.


% Predicate managing the argument concerning the incoherences
% analysis followed by the test suites generation
%
scan_args(['-filtest', Features, Mapping|Options]):-
exists_file(Features),
consult(Features), (
  exists_file(Mapping),
  consult(Mapping),
  scan_ft_options(Options), (
```

```
    groups_loaded(yes)
             ;
    not(groups_loaded(yes)),
    build_feature_groups
          ), (
    fi_output_file(FiOutput),
    open(FiOutput, write, FiStream),
    set_output(FiStream)
             ;
            not(fi_output_file(_)),
    stdout(FiStream), set_output(FiStream)
          ), (
    groups_loaded(yes), nl, time(fi_search), !
             ;
    not(groups_loaded(yes)),
    build_feature_groups, nl, time(fi_search), !
          ), (
    ts_output_file(TsOutput),
    open(TsOutput, write, TsStream),
    set_output(TsStream)
             ;
            not(ts_output_file(_)),
    stdout(TsStream), set_output(TsStream)
          ),
  fi_gen_test,
  halt
           ;
  not(exists_file(Mapping)),
  write('\t--- Invalid file: '),
  write(Mapping), nl, halt
        )
 ;
not(exists_file(Features)),
write('\t--- Invalid file: '),
write(Features), nl, halt.


% Predicate managing all other cases; the bad arguments
%
scan_args(_):-
print_usage.


% Predicates managing sub-options relatives to the Incoherences
% analysis.
```

```
%
scan_fi_options([]).

scan_fi_options(['-g', Groups|Options]):-
exists_file(Groups),
consult(Groups),
assert(groups_loaded(yes)),
scan_fi_options(Options)
 ;
not(exists_file(Groups)),
write('\t--- Invalid file: '),
write(Groups), nl, halt.

scan_fi_options(['-k', Known|Options]):-
exists_file(Known),
consult(Known),
scan_fi_options(Options)
 ;
not(exists_file(Known)),
write('\t--- Invalid file: '),
write(Known), nl, halt.

scan_fi_options(['-i', Output|Options]):-
assert(fi_output_file(Output)),
scan_fi_options(Options).

scan_fi_options(_):-
print_usage.


% Predicates managing sub-options relatives to the test generation
% for features in isolation
%
scan_iso_options([]).

scan_iso_options(['-t', Output|Options]):-
assert(iso_output_file(Output)),
scan_iso_options(Options).

scan_iso_options(_):-
print_usage.


% Predicates managing sub-options relatives to the Tests Generation
% for incoherences
```

```
%
scan_ts_options([]).

scan_ts_options(['-t', Output|Options]):-
assert(ts_output_file(Output)),
scan_ts_options(Options).

scan_ts_options(_):-
print_usage.


% Predicates managing sub-options relatives to the Incoherences
% analysis and Tests Generation
%
scan_ft_options([]).

scan_ft_options(['-g', Groups|Options]):-
exists_file(Groups),
consult(Groups),
assert(groups_loaded(yes)),
scan_ft_options(Options)
 ;
not(exists_file(Groups)),
write('\t--- Invalid file: '),
write(Groups), nl, halt.

scan_ft_options(['-k', Known|Options]):-
exists_file(Known),
consult(Known),
scan_ft_options(Options)
 ;
not(exists_file(Known)),
write('\t--- Invalid file: '),
write(Known), nl, halt.

scan_ft_options(['-i', Output|Options]):-
assert(fi_output_file(Output)),
scan_ft_options(Options).

scan_ft_options(['-t', Output|Options]):-
assert(ts_output_file(Output)),
scan_ft_options(Options).

scan_ft_options(_):-
print_usage.
```

```
% Predicate printing informations about usage of the program
%
print_usage:-
write('Usage: fi-lookup <command>'), nl,
write('    * where command must be one of the following:'), nl,
nl,
write('    -filtering <features> [options]'), nl,
write('    --> find possible incoherences'), nl,
write('\t* where options include:'), nl,
write('\t-g <file> --- feature sets to compare'), nl,
write('\t-k <file> --- list of already known incoherences'), nl,
write('\t-i <file> --- file to which reports will be written'), nl,
nl,
write('    -testiso <features> <mapping-rules> [options]'), nl,
write('    --> produce test suites for features in isolation'), nl,
write('\t* where options include:'), nl,
write('\t-t <file> --- file to which test suites will be written'), nl,
        nl,
write('    -testfi <mapping-rules> <known incoherences> [options]'), nl,
write('    --> produce test suites for given incoherences'), nl,
write('\t* where options include:'), nl,
write('\t-t <file> --- file that will contain the test scenarios'), nl,
nl,
write('    -filttest <features> <mapping-rules> [options]'), nl,
write('    --> finds possible incoherences & '),
write('produces test suites'), nl,
write('\t* where options include:'), nl,
write('\t-g <file> --- feature sets to compare'), nl,
write('\t-k <file> --- list of already known incoherences'), nl,
write('\t-i <file> --- file to which reports will be written'), nl,
write('\t-t <file> --- file to which test suites will be written'),
nl, halt.

%----------* Arguments predicate definitions *-----------------------------


                     %**********************************%


%----------* Menu predicate definitions *----------------------------------


% Predicate main, first predicate to be called when program starts
```

```
%
main:-
dynamic(contradiction_pair(_, _)),
dynamic(lfeature(_)),
dynamic(rfeature(_)),
dynamic(fi(_, _, _, _, _, _, _, _, _)),
dynamic(test_done(_, _, _)),
dynamic(iso_test_done(_)),
dynamic(fi_output_file(_)),
dynamic(iso_output_file(_)),
dynamic(ts_output_file(_)),

current_output(Stream),
assert(stdout(Stream)),
assert(groups_loaded(no)),
unix(argv(Args)), (
  ( Args = [_|[]], print_usage )
          ;
  ( Args = [_|List], scan_args(List) )
        ).

%----------* Menu predicate definitions *----------------------------------


                     %**************************************%


%----------* Rules for building the set of incoherences *------------------

% Built-in contradiction, a user can not subscribe to the same
% feature more than one time.


% Main predicates, compare all features of the left hand set to
% all features of the right hand set.
%
fi_search:-
        (
  ( groups_loaded(no), fi_search_left_to_right )
          ;
         ( groups_loaded(yes),
           fi_search_left_to_right,
           fi_search_right_to_left
         )
        ).
```

```
% Analysis of features of the left hand set againts features of the
% right hand set
%
fi_search_left_to_right:-
rule(R),
lfeature(F1),
rfeature(F2),
not(fi(R, F1, F2, _, _, _, _, _, _)),
fi_check(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),
assert(fi(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2)),
write('%---------------------------------------'),
write('---------------------------------------'), nl,
fi_infos(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),
nl,
portray_clause(fi(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2)),
nl,
write('%---------------------------------------'),
write('---------------------------------------'),
nl, nl,
write('%                             *******************'),
nl, nl,
fi_search_left_to_right, !.

fi_search_left_to_right:-
rule(R),
lfeature(F1), rfeature(F2),
(fi(R, F1, F2, _, _, _, _, _, _)), !.


% Analysis of features of the right hand set againts features of the
% left hand set
%
fi_search_right_to_left:-
rule(R),
rfeature(F1),
lfeature(F2),
not(fi(R, F1, F2, _, _, _, _, _, _)),
fi_check(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),
assert(fi(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2)),
write('%---------------------------------------'),
write('---------------------------------------'), nl,
fi_infos(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),
nl,
```

```
portray_clause(fi(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2)),
nl,
write('%----------------------------------------'),
write('---------------------------------------'),
nl, nl,
write('%                              ********************'),
nl, nl,
fi_search_right_to_left, !.

fi_search_right_to_left:-
rule(R),
rfeature(F1), lfeature(F2),
(fi(R, F1, F2, _, _, _, _, _, _)), !.

%----------* Rules for building the set of incoherences *-------------------


                     %*************************************%


%----------* Rules for incoherences identification *------------------------

% There is a contradiction between Set1 and Set2 if an element of Set1
% is opposed to another element of Set2 and vice versa.
% * The opposed elements must be defined in the feature descriptions.
%
contradiction(Set1, Set2):-
member(X, Set1),
member(Y, Set2),
(contradiction_pair(X, Y) ; contradiction_pair(Y, X)), !.


% *** RULE #d1 ***
%
% The goal of this rule is to identify  incoherences between
% two features held by the same user when no contradiction exists
% between pre-conditions, triggering events are the same and results
% are different.
%
% * Typical: Incoherence between Voice Mail and Call Waiting
%
fi_check(d1, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
not(fi(d1, [F2, Fy], [F1, Fx], _, _, _, _, _, _)),
feature([F1, Fx], PcnF1, TrgF1, ResF1), !,
feature([F2, Fy], PcnF2, TrgF2, ResF2),
```

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


% *** RULE #d2 ***
%
% The goal of this rule is to identify incoherences between
% two features held by the same user when no contradiction exists
% between pre-conditions, triggering events are the same and results
% present a contradiction.
%
% * Typical: Incoherence between Call Waiting and Incoming Call Screening
%
fi_check(d2, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
not(fi(d2, [F2, Fy], [F1, Fx], _, _, _, _, _, _)),
feature([F1, Fx], PcnF1, TrgF1, ResF1), !,
feature([F2, Fy], PcnF2, TrgF2, ResF2),

TrgF1 = TrgF2,
member(subs(U, F1), PcnF1), member(subs(U, F2), PcnF2),
not(contradiction(PcnF1, PcnF2)),
contradiction(ResF1, ResF2), !.


% *** RULE #d3 ***
%
% The goal of this rule is to identify incoherences between
% two features held by the same user when no contradiction exists
% between pre-conditions, triggering events are the same and results
% are different.
%
% * Typical: Incoherence between Voice Mail and Call Waiting
%
fi_check(d3, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
feature([F1, Fx], PcnF1, TrgF1, ResF1), !,
feature([F2, Fy], PcnF2, TrgF2, ResF2),

TrgF1 = TrgF2,
ResF1 \= ResF2,
member(subs(U1, F1), PcnF1), member(concerns(U2, F1), PcnF1),
member(subs(U2, F2), PcnF2), member(concerns(U2, F2), PcnF2),
```

```
U1 \= U2,
not(contradiction(PcnF1, PcnF2)),
not(contradiction(ResF1, ResF2)), !.


% *** RULE #d4 ***
%
% The goal of this rule is to identify incoherences between
% two features held by the same user when no contradiction exists
% between pre-conditions, triggering events are the same and results
% present a contradiction.
%
% * Typical: Incoherence between Voice Mail and Call Waiting
%
fi_check(d4, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
feature([F1, Fx], PcnF1, TrgF1, ResF1), !,
feature([F2, Fy], PcnF2, TrgF2, ResF2),

TrgF1 = TrgF2,
member(subs(U1, F1), PcnF1), member(concerns(U2, F1), PcnF1),
member(subs(U2, F2), PcnF2), member(concerns(U2, F2), PcnF2),
U1 \= U2,
not(contradiction(PcnF1, PcnF2)),
contradiction(ResF1, ResF2), !.


% *** RULE #t1 ***
%
% The goal of this rule is to identify incoherences between
% two features held by different users when no contradiction exists
% between pre-conditions, results of F1 include triggering events of F1
% and results present a contradiction.
%
% * Typical: B-(has(CFA), to(C)) & C-(has(ICS), ics_list(A)). A calls B.
% * Typical: B-(has(CFA), to(C)) & A-(has(OCS), ocs_list(C)). A calls B.
%
fi_check(t1, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
feature([F1, Fx], PcnF1, TrgF1, ResF1),
        feature([F2, Fy], PcnF2, TrgF2, ResF2),

not(contradiction(PcnF1, PcnF2)),
        subset(TrgF2, ResF1),
        contradiction(ResF1, ResF2), !.
```

```
% *** RULE #t2 ***
%
% The goal of this rule is to identify loops between features held by
% different users when no contradiction exists between pre-conditions,
% results of F1 includes triggering events of F2 and vice-versa.
%
% * Typical: B-(has(CFA), to(C)) & C-(has(CFA), to(B)). A calls B.
%
fi_check(t2, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
not(fi(d2, [F2, Fy], [F1, Fx], _, _, _, _, _, _)),
feature([F1, Fx], PcnF1, TrgF1, ResF1), !,
feature([F2, Fy], PcnF2, TrgF2, ResF2),

not(contradiction(PcnF1, PcnF2)),
subset(TrgF2, ResF1),
subset(TrgF1, ResF2), !.

%----------* Rules for incoherences identification *----------------------


                     %**********************************%


%-----------* Rules for building the set of test scenarios *--------------

% Rule generating test scenarios for each of the incoherences found
% if no analysis was previously done, it will call it to first have
% the list of incoherences, then, it will build test scenarios.
%
fi_gen_test:-
fi(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),
not(test_done(R, F1, F2)), assert(test_done(R, F1, F2)),
fi_test(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2),
fi_gen_test, !.


fi_gen_test:-
fi(R, F1, F2, _, _, _, _, _, _),
test_done(R, F1, F2), !.

% Rule generating test scenarios for the features in isolation
%
iso_gen_test:-
feature(F, P, T, R),
not(iso_test_done(F)), assert(iso_test_done(F)),
```

```
isolation_test(F, P, T, R),
iso_gen_test, !.

iso_gen_test:-
feature(F, _, _, _),
iso_test_done(F), !.

%------------* Rules for building the set of test scenarios *---------------

                     %*************************************%


%------------* Rules for deriving test scenarios *-------------------------

% Test rule to derive test scenarios for features in isolation
%
isolation_test(F1, PcnF1, TrgF1, ResF1):-
          test_header([F1, PcnF1]),
          test_pre_conditions(PcnF1, []),
          test_triggers(TrgF1),
          test_results(ResF1),
          test_footer.


% Test rule #dx, derivation of test from Rules #d1, #d2, #d3 and #d4
%
fi_test(R, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
((R = d1); (R = d2);(R = d3); (R = d4)),
% test scenario case 1, feature F1 has priority
test_header([F1, PcnF1], [F2, PcnF2], F1, R),
test_pre_conditions(PcnF1, PcnF2),
test_triggers(TrgF1),
test_results(ResF1),
test_footer,

% test scenario case 2, feature F2 has priority
test_header([F1, PcnF1], [F2, PcnF2], F2, R),
test_pre_conditions(PcnF1, PcnF2),
test_triggers(TrgF2),
test_results(ResF2),
test_footer.


% Test rule #t1, derivation of test from Rule #t1
```

```
%
fi_test(t1, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
% test scenario case 1, feature F1 has priority
test_header([F1, PcnF1], [F2, PcnF2], F1, t1),
test_pre_conditions(PcnF1, PcnF2),
test_triggers(TrgF1),
test_results(ResF1),
test_footer,

% test scenario case 2, feature F2 has priority
test_header([F1, PcnF1], [F2, PcnF2], F2, t1),
test_pre_conditions(PcnF1, PcnF2),
test_triggers(TrgF1),
test_results(TrgF2),
test_results(ResF2),
test_footer.


% Test rule #t2, derivation of test from Rule #t2
%
fi_test(t2, F1, F2, PcnF1, TrgF1, ResF1, PcnF2, _, ResF2):-
% test scenario, features enter a loop
test_header([F1, PcnF1], [F2, PcnF2], ['a', 'loop'], t2),
test_pre_conditions(PcnF1, PcnF2),
test_triggers(TrgF1),
test_results(ResF1),
test_results(ResF2),
test_results(ResF1),
test_footer.

%------------* Rules for deriving test scenarios *-------------------------


                     %*************************************%


%------------* Rules for giving informations about an incoherence ----------

% Information about the incoherence found
%
fi_infos(d1, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, _, ResF2):-
nl, write('% * Rule #d1'), write(' -> '),
write([F1, Fx]), write(' & '), write([F2, Fy]), nl,
write('%   Same user subscribed to two features having the'), nl,
write('%   same triggering events and different results'), nl, nl,
```

```
write('%      + Features pre-conditions'), nl, nl,
write('%         - Pre-conditions of '), write([F1, Fx]), nl,
writeSet('%              ', PcnF1), nl,
write('%         - Pre-conditions of '), write([F2, Fy]), nl,
writeSet('%              ', PcnF2), nl,

write('%      + Same triggering events'), nl,
writeSet('%           ', TrgF1), nl,

write('%      + Different results'), nl, nl,

write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%              ', ResF1), nl,
write('%         - Resulting events of '), write([F2, Fy]), nl,
writeSet('%              ', ResF2), nl.


% Information about the incoherence found
%
fi_infos(d2, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, _, ResF2):-
nl, write('% * Rule #d2'), write(' -> '),
write([F1, Fx]), write(' & '), write([F2, Fy]), nl,
write('%   Same user subscribed to two features having the'), nl,
write('%   same triggering events and contradictory results'), nl, nl,

write('%      + Features pre-conditions'), nl, nl,
write('%         - Pre-conditions of '), write([F1, Fx]), nl,
writeSet('%              ', PcnF1), nl,
write('%         - Pre-conditions of '), write([F2, Fy]), nl,
writeSet('%              ', PcnF2), nl,

write('%      + Same triggering events'), nl,
writeSet('%           ', TrgF1), nl,

write('%      + Contradictory results'), nl, nl,

write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%              ', ResF1), nl,
write('%         - Resulting events of '), write([F2, Fy]), nl,
writeSet('%              ', ResF2), nl.


% Information about the incoherence found
%
```

```
fi_infos(d3, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, _, ResF2):-
nl, write('% * Rule #d3'), write(' -> '),
        write([F1, Fx]), write(' & '), write([F2, Fy]), nl,
write('%   Different users subscribed to two features where'), nl,
        write('%   feature 1 concerns subscriber of feature 2, and'), nl,
        write('%   the features have the same triggering events and'), nl,
        write('%   different results'), nl, nl,

write('%      + Features pre-conditions'), nl, nl,
write('%         - Pre-conditions of '), write([F1, Fx]), nl,
writeSet('%             ', PcnF1), nl,
write('%         - Pre-conditions of '), write([F2, Fy]), nl,
writeSet('%             ', PcnF2), nl,

write('%      + Same triggering events'), nl,
writeSet('%          ', TrgF1), nl,

write('%      + Different results'), nl, nl,

write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%             ', ResF1), nl,
write('%         - Resulting events of '), write([F2, Fy]), nl,
writeSet('%             ', ResF2), nl.


% Information about the incoherence found
%
fi_infos(d4, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, _, ResF2):-
nl, write('% * Rule #d4'), write(' -> '),
        write([F1, Fx]), write(' & '), write([F2, Fy]), nl,
write('%   Different users subscribed to two features where'), nl,
        write('%   feature 1 concerns subscriber of feature 2, and'), nl,
        write('%   the features have the same triggering events and'), nl,
        write('%   contradictory results'), nl, nl,

write('%      + Features pre-conditions'), nl, nl,
write('%         - Pre-conditions of '), write([F1, Fx]), nl,
writeSet('%             ', PcnF1), nl,
write('%         - Pre-conditions of '), write([F2, Fy]), nl,
writeSet('%             ', PcnF2), nl,

write('%      + Same triggering events'), nl,
writeSet('%          ', TrgF1), nl,

write('%      + Contradictory results'), nl, nl,
```

```
write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%              ', ResF1), nl,
write('%         - Resulting events of '), write([F2, Fy]), nl,
writeSet('%              ', ResF2), nl.


% Information about the incoherence found
%
fi_infos(t1, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
nl, write('% * Rule #t1'), write(' -> '),
write([F1, Fx]), write(' & '), write([F2, Fy]), nl,
write('%   The user(s) subscribed to different features and the'), nl,
write('%   results of F1 include the triggering events of F2'), nl,
write('%   but their results are contradictory'), nl, nl,

write('%      + Different pre-conditions'), nl, nl,
write('%         - Pre-conditions of '), write([F1, Fx]), nl,
writeSet('%              ', PcnF1), nl,
write('%         - Pre-conditions of '), write([F2, Fy]), nl,
writeSet('%              ', PcnF2), nl,

write('%      + Transitivity between features'), nl, nl,

write('%         - Triggering events of '), write([F1, Fx]), nl,
writeSet('%              ', TrgF1), nl,
write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%              ', ResF1), nl,

write('%         - Including triggering events of '), write([F2, Fy]),
nl, writeSet('%              ', TrgF2), nl,

write('%      + Contradictory results'), nl, nl,

write('%         - Resulting events of '), write([F2, Fy]), nl,
writeSet('%              ', ResF2), nl,
write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%              ', ResF1), nl.


% Information about the incoherence found
%
fi_infos(t2, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2):-
nl, write('% * Rule #t2'), write(' -> '),
write([F1, Fx]), write(' & '), write([F2, Fy]), nl,
```

```
write('%   The users subscribed to different features for which'), nl,
write('%   the resulting events of F1 include triggering events'), nl,
write('%   of F2 and vice-versa, which leads to a loop.'), nl, nl,

write('%      + Different or common pre-conditions'), nl, nl,
write('%         - Pre-conditions of '), write([F1, Fx]), nl,
writeSet('%            ', PcnF1), nl,
write('%         - Pre-conditions of '), write([F2, Fy]), nl,
writeSet('%            ', PcnF2), nl,

write('%      + Transitivity between features'), nl, nl,

write('%         - Resulting events of '), write([F1, Fx]), nl,
writeSet('%            ', ResF1), nl,
write('%         - Including triggering events of '), write([F2, Fy]),
nl, writeSet('%            ', TrgF2), nl,

write('%      + Reverse transitivity, leading to a loop'), nl, nl,
write('%         - Resulting events of '), write([F2, Fy]), nl,
writeSet('%            ', ResF2), nl,
write('%         - Including triggering events of '), write([F1, Fx]),
nl, writeSet('%            ', TrgF1), nl.

%-----------* Rules for giving informations about an incoherence ----------
```