Runtime Resolution of Feature Interactions in Evolving Telecommunications Systems

Stephan Reiff-Marganiec

A Dissertation submitted to the University of Glasgow in partial fulfillment of the regulations for the degree of Doctor of Philosophy

May 2002

Department of Computing Science
University of Glasgow
Glasgow, UK

Abstract

Feature interactions in telecommunications is an active research area. Many approaches to solve the so-called *feature interaction problem* have been proposed. However, all these approaches consider feature interaction as a somewhat isolated problem, in particular it is not seen in the context of *evolving legacy systems* and third party features in a *deregulated market environment*. An exception is the approach by Marples and Magill [MM98, Mar00], which presents an interaction *detection* mechanism and an essentially *manual resolution* approach.

We develop an *automatic resolution* approach that can be integrated with Marples and Magill's detection mechanism. We distinguish two key concepts, namely solutions and resolutions. The former are essentially possible behaviours of the system, they are not qualified as desirable or undesirable, the latter are the desirable solutions. Our approach allows for automatic removal of undesired behaviour and selection of the "best" desired behaviour.

The correctness, complexity and suitability of our approach are analysed. Two case studies support these more theoretical considerations.

Our approach is transferable to other areas, such as quality of service management, and is not restricted to network architectures with a single point of control.

Declaration

The studies outlined in this thesis were undertaken in the Department of Computing Science, University of Glasgow, under the supervision of Professor Muffy Calder. This dissertation has not been submitted at any other university. All of the work was performed by the author, except where otherwise indicated.

Work described in some sections has been previously published, in particular: the idea of the hybrid approach (Chapter 4) has been published in [CR00], the specification of the solution space (Sections 5.1 to 5.5) is published as [Rei00], the running example (Chapter 3) was developed in the context of the feature interaction contest [KMMR00] and initial ideas for resolution rules (Section 6.2) have been formulated in [CMRT02].

Material presented in Chapters 5, 6 and 7 is developed solely by the author.

Stephan Reiff-Marganiec
Glasgow, April 2002

Although the telegraph, together with the ensuing telecommunications revolution, came in the nineteenth century, its origins can be traced all the way back to 1753. An anonymous letter in a Scottish magazine described how a message could be sent across large distances by connecting the sender and receiver with 26 cables, one for each letter of the alphabet. The sender could then spell out the message by sending pulses of electricity along each wire. For example, to spell out `hello` the sender would begin by sending a signal down the `h` wire, then down the `e` wire and so on. The receiver would somehow sense the electrical current emerging from each wire and read the message. However, this 'expeditious method of conveying intelligence', as the inventor called it, was never constructed, because there were technical obstacles that had to be overcome.

Simon Singh, The Code Book (p. 60)

Acknowledgements

I wish to thank Prof. Muffy Calder for many stimulating discussions and for always querying explanations of key concepts until I expressed them in a precise form. Muffy's guidance throughout this work proved invaluable and without her this thesis would not exist.

EPSRC is to be thanked for funding the Hybrid Feature Interaction Project (GR/M03429/01) which provided my financial support over the past three years. In the framework of this project, I had extensive contact with Prof. Evan Magill, Dave Marples and Mario Kolberg – thank you for the time spent discussing technical aspects.

The work was not conducted in isolation, and many other people have provided me with valuable thoughts. I want to thank my fellow students and the staff in the Department of Computing Science here at Glasgow for providing a stimulating research environment. Special thanks are due to Dr Alice Miller for completing the arduous task of proof reading a draft of this thesis.

Finally, I wish to express thanks to my wife for her patience when I was only talking about telephone switching systems for days on end.

Thank you!

Stephan

Contents

List of Figures

Chapter 1

Introduction

1.1 Background to Research

Three intertwined problems motivate the research presented in this dissertation. The *Feature Interaction Problem*, *Legacy Systems* and *Deregulation of the Telecommunications Market* each pose interesting problems on their own, in telecommunications switching software the three areas combine to form a challenging problem.

Features provide additional functionality to a basic service. In the telecommunications setting this could be a *call waiting* facility, and in an automobile industry setting an *engine immobiliser*. Packaged software might have features such as an *equation editor* or a *movie playback plug-in*.

With the presence of multiple features in a single system, it is not unlikely that they affect each others functionality; when they do so we encounter a feature interaction. These interactions can provide useful behaviour, and thus be desired. However, if interactions lead to behaviour that is inconsistent with the user expectations or even to breakage of the system, then they are clearly undesired.

The feature interaction problem in telecommunication systems was recognised to be important by industry and academia towards the end of the 1980s and since then an active feature interaction community has developed. The ongoing work is best represented by the series of International Feature Interaction in Telecommunications Workshops [VGL92, BV94, CO95, DBL97, KB98, CM00].

Traditionally, feature interactions were resolved at design time – experienced designers were able to identify interactions using pragmatic approaches. This was made possible by the rather small number of features and the fact that most features were produced by the same company that developed the respective base product. Often features were integrated by simply adding to the code of the original system.

A change in user expectations and the market situation has led to the requirement for more features. Users expect more features, and operators want to provide more features as base services are becoming less profitable. Thus, the number of developed features is growing rapidly and a quick time-to-market is required to maximise profits, both of which are proving challenging to the traditional, pragmatic approach of integrating features.

In addition to the growing number of features, two further complications have been recognised thus rendering the pragmatic approaches even less useful: *fragility of legacy code* and *multi vendor environments.*

The switching software in modern telecommunications switches – also called stored program control switches (SPC)[RV94] – is large, often insufficiently documented (or the documentation is out-dated) and generally fragile. The software may have evolved over a period of years, even decades. The fragility and poor documentation makes the current approach of simply integrating features at a source code level undesirable (independent of the feature interaction problem). However, the external interaction with the legacy system can be assumed to be well documented or understood, thus it is known which kind of events lead to which responses from the system.

Deregulation of the market introduces two problems. On one hand a highly competitive multi-vendor environment requires quick time to market of new developments to maximise profits. In addition, features stemming from different vendors must co-exist. Resolving interactions at design-time becomes impossible, as on one hand the source code and documentation of competitors features are unavailable for proprietary reasons and on the other hand an operator might encounter new features at runtime only.

In the telecommunications domain we encounter a combination of market deregulation and large, evolving legacy systems together with feature interactions. A solution to the feature interaction problem in this domain must take diverse factors into account: documentation of legacy systems cannot be trusted and implementation details of third party features are unknown. Neither legacy nor third party features can be changed; they are, however, required to work together. Quick time to market complicates elaborate testing in the context of an ever growing number of features, especially when several vendors introduce new features simultaneously.

Despite a large body of work existing in the feature interaction area, little attention has been given to resolution techniques. In the offline work, resolution of found interactions is achieved by redesign, and thus is a smaller issue. However, in the online context, resolution is crucial. There is currently no method that can resolve interactions at runtime without requiring human input at runtime or predefined tables of interaction scenarios with their respective resolutions. The exception is work based on feature negotiation which requires new architectures and hence is not practical in a legacy context.

As feature interaction poses a problem for the development of new services and both new services and legacy systems are in use in a deregulated market a resolution method that is suited to this context is required. We show the feasibility of such a method.

1.2 Research Questions

The overall aim of our approach is to detect and resolve feature interactions in evolving telecommunications systems. It is useful to identify the particular problems that we need to overcome.

Aims

Detect and resolve feature interactions

- in the presence of large legacy systems
  - no reliable documentation
  - fragile code
- in a deregulated market
  - no design time information about third party features
  - short development periods
  - the presence of other features might only be recognised at runtime

We develop an approach to resolve and detect feature interactions in the context of evolving legacy and third party systems addressing the above aims with a number of objectives.

Objectives

The approach shall

- be embeddable in legacy and new architectures
- not require changes to features or legacy code
- not require design information of features or legacy code
- automatically detect and resolve interactions at runtime

The last objective ensures that features can be developed quickly. Individual features can be developed separately and only the under the constraints possible testing and interaction avoidance techniques need to be performed. The runtime approach resolves any arising (and remaining) interactions with all features from the same and other vendors.

To fulfil the objectives we envision a process that first quantifies potential solutions and second identifies the "best" such solution. The notion of quantifying potential solutions leads to questions such as:

- What is a solution?
- How many solutions are there?
- How do we find them all?
- How can solutions be found at runtime assuming only observable behaviour of features?

We are concerned with selecting one of the possible solutions – preferably the best. Again, we need to answer a number of questions:

- When is a solution good/bad?
- What are acceptable solutions?
- When can we say a solution is better than another (assuming both are good)?
- Does a partial order based on quality of solutions exist?
- How can we describe bad behaviour in a general way?
- What information is required, and do messages contain the relevant information?
- How can good solutions be extracted from the solution space at runtime?

Finally, the analysis of the applicability raises the questions:

- When can we resolve interactions and when not; and if not, why not?
- How important is the system architecture?
- Is the complexity of the approach acceptable for a runtime environment?
- How well does the approach scale?
- What is the scope of applicability? Is it transferable? Will it work with new features?
- How effective is the approach?

We answer all these questions, though some answers are more in depth than others. Clear answers are provided to the questions about solutions and good solutions. Though, we find that a numeric answer as to how many solutions exist is not possible for the general case and finding all possible solutions might not be feasible at runtime – though we also show that if it is not feasible it is not required. The applicability questions are also answered, though scalability and complexity would benefit a further analysis in an operational system.

In this dissertation we show:

Thesis Statement

An automatic runtime approach to resolve feature interactions in multiple point of call control environments in the presence of legacy and third party features is desirable. We demonstrate the feasibility of such an approach using a transactional approach.

1.3 Outline of Report

In the next chapter we give a more detailed description of the research context. In particular we concentrate on the Transactional Approach presented by Marples and Magill [MM98, Mar00], as this forms the basis for our work. Chapter 3 introduces a running example – a basic call model and a set of 14 features. We have advocated a hybrid approach [CMM99, CR00] which will be discussed in detail in Chapter 4.

Having provided a detailed background, we then proceed to consider the main research issues. We specify and implement a method of detecting interactions and computing all possible solutions to a detected interaction (Chapter 5). This builds on Marples and Magill's work, providing several extensions. In Chapter 6 we define novel methods for resolving detected feature interactions. Chapter 7 explores issues of correctness and complexity of the presented approach. An empirical evaluation of two sets of features leads to a discussion on success and suitability – a critical evaluation of the approach.

Finally our main conclusions are summarised in Chapter 8. We also consider the transferability of the presented approach to areas outwith feature interaction in telecommunications, and how new developments and the paradigm shift to *Voice over IP* impacts on the whole research area.

Two appendices provide detailed description of the features from the running example (Appendix A) and the Haskell source code of the implementation (Appendix B).

1.4 Basic Terminology

Throughout this document we use a number of terms, they are also used in the relevant literature. Here we briefly introduce them and discuss how they are used.

We use the term **interaction** to mean that two or more features are active at the same time because they either trigger each other or react to the same external trigger. When referring to interactions, no judgement is made as to whether they are desired or undesired.

In the literature the terms *interaction*, *interworking* and *interference* are sometimes used. Most authors provide their own definition and thus the terms might refer to the same concept or to different ones depending on the author (i.e. some authors use different terms to qualify between desirable and undesirable interactions, while others simply use all three terms interchangeable).

Service and *feature* are sometimes used interchangeably. The IN standard [ITU93b] distinguishes the two as follows: a features is additional behaviour (network capability) provided by the switching system and a service is a combination of features provided by the operator to the customer. For our purpose the difference is irrelevant and we will simply use the term **feature** to describe the components adding behaviour to the system.

We use the terms *online* and *runtime* interchangeably.

1.5 Delimitations of Scope and Key Assumptions

Our aim is to develop a resolution mechanism for feature interactions in the specific context of legacy systems and third party components. The key assumptions and delimitations are such that the method is realistic for this setting.

We assume that the interacting components, called features, are either third party or legacy software – thus the approach does not make use of internal state or other internal information of these features. Rather it concentrates on the exchanged messages at the interface, i.e. the observable behaviour. For our purpose, a feature is simply a black box that receives messages and might respond with messages (it could simply consume input without an observable reaction).

Message passing is the communication mechanism between parts of the system. In telecommunications systems this is natural, but we also find that other component based systems communicate with event (or message) exchange. However, the approach does not depend this. What is crucial is that the communication between the components can be intercepted, delayed and blocked.

We must be able to introduce a new component, the feature manager, into the system between the features and the environment. In the standard for advanced intelligent

networks [ITU93b] this component is already included in the architecture. However, in non-IN systems, which legacy systems often are, a new system component must be incorporated. As the feature manager must be able to intercept all messages between the features and the environment and block these whenever required, it only requires to interface the present components. Thus the components do not need to be changed, it is the communication path that must be routed through the feature manager.

The transactional approach we adopt requires a way of resetting a part of the system to an earlier state. The part concerned involves the features and all global variables that are affected. It is assumed that the features do not have any side effects (other than changing certain well known global variables, such as the status of the phone line). There are several ways to restore an earlier state of the system: we can start new instances and rollback to previous ones or we can rollback to an initial state and replay messages – the latter being very similar to techniques used for error recovery in database systems. These are just two possibilities, though others might be thought of. The system must allow for at least one such method.

We can imagine features that require further user input (in fact the Teenline feature that we introduce later requires the user to enter a PIN number). However, we assume that no feature can gain control over the system to the extend were it can block other features receiving messages and responding to the same.

Although we do not expect to know details of the internal working of each feature, we require some understanding of the ontology of messages. We can place our knowledge on a scale ranging from nothing to knowing all detail. However, knowing nothing is not very realistic, as we are aware of the structure of the message (in the case of the features a message contains an event and maybe a parameter). As we are able to interact with the feature we also know the message set, and that messages have a consistent meaning. For each message the semantics is also known, as otherwise responses from the system would be meaningless.

The success of the resolution depends on the available information. Thus, the richer the information in the exchanged messages, the better the resolutions will be. However, we show that even relatively minimal information will permit adequate resolution.

In summary, we assume a system where communication between components (such as features and legacy systems) can be intercepted, delayed and blocked. The internal behaviour of components is not of interest to us. Knowledge of the semantics of the exchanged messages improves the quality of the resolutions but is not strictly required.

1.6 Contributions of Dissertation

The main contribution of this dissertation is an automatic, runtime approach to **resolve** detected feature interactions in telecommunications systems. The approach addresses problems in conjunction with the existence of large, evolving legacy systems in a deregulated market environment. We show the technical details of such an

approach and analyse the requirements and success in case studies, thus proving the desirability and feasibility.

Despite the work being primarily conducted in the context of telecommunications systems and thus concentrating on features of telecommunications switching systems, the approach does not depend fundamentally on domain specific assumptions. The approach has been transferred to interactions of quality of service management features in a multimedia context [BR01].

The approach allows resolution of detected interactions in telecommunications systems at runtime. We require only minimal information about the features in the system, namely an understanding of the exchanged messages. Thus the approach is applicable in a legacy context where details of the implementation are often not available, and also in a multi-vendor environment where the internal working of components is unavailable.

This work is novel, because it tackles the problem of **resolving** feature interactions **in the context of legacy systems and a deregulated market**. There are two related approaches Marples and Magill [MM98, Mar00] and Buhr et al. [BAE+98]. We distinguish our approach in the following way.

Marples and Magill's work, which we build upon, is restricted to single point of call control settings that are realistic for PBXs, but not in public networks. Furthermore, they only provide a crude resolution mechanism as their work concentrated on detection. However, we show that it is possible to extend the approach in a form that allows for completely *automatic* resolution based on a theory of interactions. Our approach allows for multiple point of call control settings.

Buhr et al. [BAE+98] use a blackboard via which features can negotiate a resolution. However, the blackboard approach requires a new system architecture. In the context of large legacy systems this is not suitable. In their approach features are represented by agents which interact to achieve the goals of the features – this is generally not applicable in legacy systems. Our method, on the other hand works in *existing legacy architectures.*

1.7 Summary

We have presented an overview of the background of the research and of the research questions that we need to address. In addition we have provided an outline of the report and discussed the basic terminology that we will use and the key assumptions that we make. Thus, we have provided the necessary vocabulary and setting for the research.

The thesis presents an automatic runtime approach for the detection and resolution of feature interactions in telecommunications systems. The work builds on the ideas of Marples and Magill, but is clearly novel and distinct. A clear advantage over

other promising runtime approaches (e.g. using blackboards) is that we complement an existing architecture rather than requiring fundamental architectural changes of the system to be made.

Chapter 2

Background

2.1 Introduction

In Chapter 1 we briefly presented the feature interaction problem and highlighted that many approaches have been developed to tackle it. We now present the background and the problem in more detail.

We argue that most approaches are unsuitable for an evolving legacy system – one of our targets is to support such systems. Our other target, multi vendor environments, requires interactions to be detected and resolved at runtime, thus we will concentrate on the proposed runtime approaches. Our work extends the transactional approach of Marples and Magill, hence this will be discussed in some detail.

2.2 General Literature on FI Detection and Resolution

2.2.1 Telephone Switching Systems

The simplest telephone network has two users connected by a wire and is often called the "tin-cans-connected-by-wire" model. When more users are in the network, they could have such connections to each other. Replacing the multiple handsets (cans) at each terminating end would introduce the need for a switching facility (at the users end). This system would need $N(N-1)/2$ two-way wires to connect N users.

Reducing the amount of wiring required is achieved by concentrating the switches in a local exchange – hence we obtain what is commonly called a telephone switch (or telephone exchange). Several of these switches can be interconnected, in the same fashion as telephones are connected to the local exchange. We concentrate on the internal workings of these exchanges, encapsulated in the term *telephone switching systems.*

The lines (wires) from the subscribers terminate in SLTU's (subscriber line termination units), which form one side of the main distribution frame (MDF). The other side is formed by switching equipment. Jumpers physically connect SLTU's to the switching equipment, thus simplifying reconfiguration. Both the users' lines and the equipment are numbered (directory numbers and equipment numbers). The relation between these is either stored in logic circuits (electromagnetical switches) or in databases (stored program control switches).

The development of stored program control switches (SPC) started in the early 1960s, with a coupling of telephony and electronic circuits and computers. The 1ESS (AT&T Bell Labs, 1965 [KKV64]) was the first public SPC switch. SPC switches have mostly replaced electromagnetical ones, and offer many advantages. The most important advantage for our work is that they allow the addition of subscriber facilities or, in our terms, features.

Switching itself is a challenging engineering problem; we distinguish between packet and line switched networks. Computer networks are usually packet switched, telephone networks line switched (although developments like *Voice over IP* deliver standard telephone services over packet switched networks). Data can be split easily into smaller chunks and these chunks can be delivered within certain time intervals, which naturally fits packet switching. Speech on the other hand is continuous and problems occur as variations in the delivery process (e.g. delays) complicate reconstruction (of split data). Nevertheless, we will not consider these issues in further detail here, because current features in telecommunications systems are not concerned with quality of service.

Stored Program Switches

Stored program switches are described extensively in [RV94]; we only consider the aspects relevant to our work here.

The SLTUs are scanned in regular intervals for incoming signals (such as *offhook*), detected signals are placed in a queue and passed on to the call processing programs. The processing depends on the signal, e.g. an *onhook* requires the call to be cleared down, all connected equipment (memory resources or physical connections to signal generators) to be released and data for billing to be gathered. *Offhook* signals require a dial-tone to be generated and connected to the respective line and memory to be allocated for the processing.

In order to store data about the users, two kinds of record are kept in memory: the call record and the subscribers record. The former is used to store call specific data, such as start and end time, current state of the call process and the switch path. The latter contains subscriber specific data. It is in here that the equipment numbers and directory numbers are related, and the class of service record (which is a part of the subscriber record rather than an entity of its own) is stored.

The class of service record consists of transient and semi-permanent data. Transient data can be changed by the user or during a call, semi-permanent data only by the network operator. Typical transient information is the line status (e.g. busy, engaged) or the activation status of features (e.g. *call forwarding* is activated to forward calls to number 456). The line type (e.g. domestic, business, ISDN) or a barring level (e.g. no international calls, no outgoing calls or no incoming calls) as well as information regarding subscription to features (the user can subscribe to a feature, whether it is activated or not by the user is not an issue for the operator, i.e. the user might still get charged) is stored in the semi-permanent data.

It should be noted that these records are used and accessed by the basic call software and features that are integrated with the basic call. These records shall not be accessed by third party features, and use might be restricted for new (proprietary) features in general.

2.2.2 Feature Interaction

Recall that features provide additional functionality to a basic service. Typical examples for telecommunications systems are *call waiting* or *three way calling*. The basic service in telecommunications is often called POTS (Plain Old Telephone Service). POTS provides basic connectivity between two users which is then extended by features. For example, the *call waiting* feature provides the capability that the subscriber can be notified of an incoming call and then toggle between two calls.

The notion of *feature interaction problem* was introduced for the first time at the seventh International Conference on Software Engineering for Telecommunications Systems by Bowen et al. [BDC+89]. Since then many researchers in industry and academia have considered the problem. The results of this research have been mainly reported in the series of proceedings from the International Workshop on Feature Interactions in Telecommunications Systems (the first of which was not formally published), [VGL92, BV94, CO95, DBL97, KB98, CM00].

As pointed out by Velthuijsen et al. [VGL92], feature interactions are not a problem restricted to telecommunications software, but apply to all large software systems that require constant upgrading. Furthermore, they claimed that "the feature interaction problem has been a major obstacle to the rapid development of new telephone services", which provided a strong motivation for research in the area by highlighting the importance of the problem.

2.2.3 The Problem

The problem can be expressed in a simple way, considering features and their interworking: each feature on its own works as expected, but two (or more) together do not function as specified. Thus, when considering interactions we make the important assumption that individual features work correctly.

Cameron et al. [CGL+94] defined feature interactions as:

Definition 2.2.1 (Feature Interaction) All interactions that interfere with the desired operation of the feature and that occur between a feature and its environment, including other features or other instances of the same feature. Additionally, interference of one part of a feature with another part of that feature (e.g. in the case of distributed implementation of a feature) is considered to be a feature interaction.

It is important to point out that feature interactions can be desired and undesired: sometimes we want a feature to change the behaviour of another one, in which case we have a desired interaction. The problematic interactions are the undesirable ones, as these might lead to anything from user annoyance to a breakdown of the network.

Originally, this problem tended to exist within a single organisation providing services. Hence the problem could be coped with due to a relatively small number of features and knowledge of coding details of each feature. Interactions were detected manually. Changes in the telecommunications industry complicate the problem for the following reasons:

- the industry is developing from a monopolised situation to a multi-vendor environment due to market deregulation,
- the number of features required and provided is growing rapidly,
- changes in the network architectures, e.g. IN (intelligent network), make it easier to provide features,
- call control is becoming more distributed, geographically (a call might be controlled by several switches) as well as through a stricter separation of hardware and software (and ever more consistent modularisation of the latter),
- legacy systems (e.g. operational code and hardware) have been so expensive in their development and deployment that the only option is to incorporate them.

2.2.4 Related Work

Most published work on feature interaction refers to the POTS (Plain Old Telephone System) or Intelligent Network (IN) ([ITU92, ITU93a, ITU97]) context. However, Tsang and Magill ([TMK97]) consider feature interactions in broadband networks and, more recently, several contributions to the last Feature Interaction Workshop [Hal00, LS00, ZJ00, BP00] discuss feature interactions in other domains such as e-mail, mobile services and internet telephony.

We assess some of the existing approaches. A particular interesting paper is the Feature Interaction Benchmark [CGL+94], which is one of the most cited papers in the area, although the authors believe it to be out-dated by now [1].

Taxonomy of Feature Interactions

Cameron et al. [CGL+94] developed a benchmark to support classification of interactions and to judge the coverage of detection and resolution approaches. They suggest two main categorisations: nature of interactions, and cause of interactions.

[1] Personal communications with the authors

As the benchmark is used widely, we briefly outline the main points. Note that the benchmark does not classify solutions, but rather attempts to provide a partitioning of the problem.

In the context of interactions, three dimensions have been identified:

1. Kind of feature involved in the interaction (customer features – system features)
2. Number of users involved in the interaction (single user – multiple user)
3. Number of network components involved in the interaction (single component – multiple components)

Customer features are those that a customer can use, e.g. *call waiting* or *call forwarding*, whereas system features are those concerned with administration and operation (e.g. billing). Single user interactions are those where features subscribed to by one user interact (e.g. *call waiting* and *call forwarding on busy*). Multiple user interactions are those with interactions between features subscribed to by different users (e.g. *call forwarding on busy* and *terminating call screening*). Single component interactions arise when features at one network node (or component) interact, multiple component interactions arise when multiple network nodes are involved.

This benchmark paper concentrates mainly on customer features, as the authors claim that system features cause fewer interactions. Five categories of interactions are explored:

SUSC (Single-User-Single-Component) interactions: These occur because one user subscribes to incompatible features on the same network node. They occur in two flavours: functional ambiguities (different features handle the same situation differently) and one feature hindering another in its proper execution.

SUMC (Single-User-Multiple-Component) interactions: These occur because the features deployed in one node of the network are unaware of those in a different component, but the user's services are provided by both nodes.

MUSC (Multi-User-Single-Component) interactions: These occur when several people share a physical phone line and hence the subscribed features. Users might set up two contradictory features like call forwarding to a number and originating call screening to the same number.

MUMC (Multi-User-Multi-Component) interactions: These occur when users subscribe to features provided on different nodes and those features are not compatible with each other.

CUSY (Customer-System) interactions: These can occur at points where administration and user features are activated simultaneously. Probably the largest area in this class is interactions with billing features.

A number of causes of interactions can be identified. These causes fall into three groups:

Violation of feature assumptions: The development of a feature relies on certain assumptions, like availability of data, call control or the signalling protocol. Over time these assumptions may lose their validity and hence the feature can no longer operate properly.

Limited network support: A network is a physical entity and as such restricted. The most obvious restriction is bandwidth shortage, leading to competition between features and hence to interaction.

General Problems of distributed systems: Many problems of distributed systems are well known, for example race conditions and/or resource contention.

We can see that interactions occur at different levels in the system and have multiple causes. However, there is no claim that the above list is exhaustive. A complete solution would need to address all those problems and also consider those which additionally might occur.

Some of the examples provided by Cameron et al. might in fact not be interactions. For example the dialling of an 0800 number from outside the country being not possible is not an interaction, it is merely a defined delimitation. Multiple users using different (and maybe contradicting) features on the same physical telephone is also not an interaction – at least as long as telephone systems cannot distinguish user identities other than by equipment numbers .

The features analysed by Cameron et al. are solely concerned with call control and billing in an IN architecture. A current taxonomy would need to take many more aspects into account.

We will return to the introduced interaction categories. As for the causes of interactions, solutions can be achieved at a different level. For example, there is much active work to handle intricacies of distributed systems. The restriction imposed by limited network support loose importance in the context of new communications architectures with rich communications protocols and more flexible terminals.

A New Taxonomy

As we have shown, the taxonomy provided by Cameron et al. [CGL+94] is rather dated. We consider the following questions and issues crucial when attempting to classify features and feature interactions:

1. Shared trigger vs. sequential action: Are features triggered by the same event or does one feature's response lead to the triggering of another feature?

2. Technical interaction vs. user intention violation: Is the interaction caused by the features dictating inconsistent reactions in a given state or simply by the unfulfillment of a user expectation?

3. Subscriber vs. non subscriber: e.g. forwarding affects both the subscriber and the caller (who is not the subscriber), whereas outgoing call barring only affects the subscriber.

4. Single vs. multiple points of call control: In some architectures calls are controlled centrally, in others not, which has an impact on feature interaction handling.

5. Nature of features: Features can be grouped by behaviour, e.g. call control, billing, quality of service or management features.

6. Call control behaviour: e.g. all forwarding features are similar in some way, conferencing features are again similar.

7. Inter vs. intra portfolio: Do features belong to the same provider or do they belong to different providers?

Due to the fast evolution of the telecommunications area it is impossible to claim that this set of questions is or will remain complete. But, we assert that this taxonomy allows us to pitch our work in the area. We address technical interactions in both single and multi point of call control systems. Our approach works for detection and resolution of both shared trigger and sequential action interactions.

Technical interactions are similar to Type I interactions as defined by Hall [Hal98]. Type I interactions are classified to be those where two features expect the system to move into inconsistent states or to exhibit inconsistent observable actions. In contrast Type II occurs, in the absence of a Type I interaction, when one feature disrupts the state properties that another has assumed. Type III interactions occur in the absence of Type I and Type II when the system fails to meet the user requirements, and thus compare to user intention violations.

The features that we have investigated are either billing or call control – currently quality of service and management features are only emerging. In Chapter 3 we give a running example and give categories of features. All features within a category are similar enough to be handled in the same fashion by detection and resolution approaches, hence the rather select set of features in the running example does indeed represent a much larger set of features as can be found for example on Lucent's PathStar switch [Luc].

For the purpose of this thesis we assume that we deal with an inter provider portfolio of features, as this group is more challenging. Our method will also apply to intra provider portfolios, but better methods can be found there, as more information is available (which can be used to detect and resolve interactions during design time).

Approaches to the Feature Interaction Problem

Several authors have classified approaches according to certain criteria. We will not discuss these classifications (there is no additional gain in showing how the body of work can be ordered), but rather concentrate on the techniques developed to solve the feature interaction problem. For the purpose of this section we distinguish approaches that identify a service engineering process, those that use formal reasoning to improve certain phases of the development process (usually design) and those that attempt to detect and resolve feature interactions at runtime.

Service Engineering Approaches

Service engineering uses techniques developed in the general context of software engineering. The techniques are mostly applicable at the requirements or specification phase of a general software engineering process.

Notably, studies using use case maps (or models; UCM) [NKHL00, ACC+00, KS94] have been applied on industrial scale systems to obtain results about their usefulness. Other, more academic, techniques involve specifications in some formal notation, where static analysis of the composition of the specified features enables feature interaction detection [Tur98, Pre97, KCK+95].

Many of the approaches in this category provide automatic approaches for filtering [KKM94, NKHL00, HS98, Bre00], that is identifying scenarios which might be prone to interactions. These cases then need to be analysed using various techniques, either automatically or manually. Nakamura et al. [NKHL00] show that about half of all possible interaction scenarios can be removed using their filtering technique because no interaction will occur. Performing more detailed analysis upon the remaining scenarios, they show that about 42% lead to interactions.

Once interactions have been detected at the specification stage, the feature specification can be adapted to remove any occurring interactions. As the detection process is automatic (or at least semi-automatic) there is the potential to cope with a growing number of features. However, as the approaches target the specification and requirements level, they are only applicable to known feature sets – i.e. those features where the relevant details are available when the process is applied. While the approaches allow for features of one vendor to interwork as expected, they are unsatisfactory in a multi-vendor environment where many details are not available and new features are often only encountered at runtime.

Some approaches [KK00, Kim97] consider service engineering from a management perspective, thus taking into account that a deregulated market poses problems that are disjunct from the pure technical issues. These approaches are interesting in their own right, but are less relevant here, as we concentrate on the technical side of the problem.

Formal Reasoning Approaches

Formalisms can be used for specification, as discussed above, but many approaches go further by using formal reasoning techniques. Formal reasoning is based on models of the system, so an immediate drawback is that the models might not accurately represent the implementations. However, interesting results have been achieved – mainly with respect to a better understanding of the problem. We can distinguish three groups of approaches: those specifying behaviour of features as formal models, using automata or transition systems, those specifying properties formally, using a logic, and those using a combination of both.

Property only approaches [BJK94, RH97, FN00] usually apply theorem proving or model checking techniques to identify inconsistency or unsatisfiability of properties once they are combined.

Behaviour only specifications are often used with specifically developed tools, or if standard languages such as CSP or LOTOS are used, the existing tools are employed. Analysis resulting in e.g. deadlock, non-reachability or non-determinism means that a feature interaction exists. Examples of such approaches are [Tho97, AA97, YO98, KB00].

Combining models of behaviour with specified properties is used in several approaches [SL95, Gib97, Tho97, PR98, KL98, dORZ98]. All these approaches use the same definition of feature interaction: if feature $F_1 \models \phi_1$ and $F_2 \models \phi_2$ but $F_1 \oplus F_2 \not\models \phi_1 \wedge \phi_2$, then an interaction exists (ϕ_1 and ϕ_2 are properties). Common to the approaches is the use of standard tools for simulation or model checking. However, the state space explosion problem generally does not allow for complete analysis. Recently Calder and Miller [CM01] succeeded in fully model checking systems of four users with two features using the model checker SPIN.

Discussion: Service Engineering and Formal Reasoning Approaches

The formal reasoning based approaches have two major drawbacks, one being that the state space explosion problem generally disallows complete assurance of detection of all interactions and the other being that the approaches are applied to models of the features. The latter requires a good understanding of the available features and an awareness of which features will be available. However, as discussed earlier, this is often not possible. Furthermore, models are abstractions of the real system. In the abstraction process, new interactions could be introduced as well as existing ones removed. The approaches prove very useful, for testing developed features for possible interaction, but fail in multi-vendor environments.

The service engineering and formal reasoning approaches are not only less applicable in multi-vendor settings, but are unsuitable within a legacy systems context. Legacy systems have already been developed (thus changing specifications is not possible). Their evolution over time means that specification details, as well as properties

extracted from the documentation, might be incorrect. In both classes of approaches, resolution occurs as redesign or changes to the requirements, which clearly is not possible in a legacy system.

Runtime Approaches

Runtime approaches attempt to overcome the problem posed by having multiple vendors by proposing ways of detecting and resolving interactions at runtime. This has the advantage that the real implementation of features can be used, rather than abstractions and models of them. Further, the features exist in their natural environment. Clearly this allows for quick development of features, as each feature can be designed separately. Interaction handling is removed from the development stage, the runtime approach handles possible interactions. However, it is still desirable to apply design time techniques whenever possible, rather than completely relying on runtime approaches.

A number of runtime approaches have been developed, and we can broadly distinguish three classes: one and two phase *feature managers* and *negotiation* based approaches. Note that one and two phase only reflects the number of distinct *online* phases, most of the one phase approaches require some information from an offline phase.

Feature Manager – One Phase Approaches. A basic feature manager is defined by the ITU-T standard for Intelligent Networks [ITU93a]. However this only prevents multiple instances of IN and non-IN services being active in the same call segment. More advanced approaches use feature managers to detect and resolve interactions. By using information about the features and their potential interactions multiple features can be allowed to be active across the call (in one or more call segments).

Homayoon and Sing [HS88] propose such an approach, whereby the feature manager is provided with a number of tables describing relations between two features. The status of one of the features is examined and then the activation or use of the second is allowed or disallowed according to its relation to the first feature. Similarly, Cain [Cai92] proposes a feature manager that only passes events to features that are known to be non-interacting using information contained in tables. Activation is not considered.

Fritsche [Fri95] determines at runtime which features are – in their term – "interested" in a proposed event. A specification of the features is provided to the feature manager in the form of roles, i.e. a feature's impact on a device. Features effect changes to devices and the feature manager observes whether roles are violated (an interaction). Interactions thus found are resolved by a predefined resolution matrix.

All these approaches require data about interaction and resolution in the form of tables to be provided to the feature manager – thus they are not useful in the context of legacy and third party systems. Marples and Magill [MM98] propose a feature manager approach that does not require a priori data, but assumes an interaction to have occurred when more than one feature intends to handle an event. They then use

a *rollback and commit* algorithm to determine possible resolutions at runtime. Their resolution mechanism is a simple precedence scheme. As our work is based on this approach, we will explore it in more detail in the next section. The approach was initially presented by Marples and Magill [MM98] and was further refined by Marples [Mar00]. For the remainder of this document we refer only to the work of Marples [Mar00] as this provides the more comprehensive reference.

Feature Manager – Two Phase Approaches. The one phase approaches described above require data about the features and their potential interaction acquired during an offline stage, the exception being [MM98]. However, some two phase approaches circumvent this requirement.

Aggoun and Combes [AC97] propose a "pre-deployment" phase where a passive observer gathers information about the feature behaviour in the network. The gathered information is then used by the active observer (essentially a feature manager) in the operation phase of the service to detect and resolve interactions. Similarly, Tsang and Magill [TM97] gather behaviour "signatures" of features in an isolated online environment (with just the feature under observation being active) and store these in a database. The feature manager then accesses this database during public network operation to detect and resolve interactions.

Both approaches require features to be executed in an isolated environment, which might not always be practical. Moreover, it is impossible to gather information about features at other network nodes that might interact when the respective users are involved in a call.

Negotiation Approaches. A markedly different alternative is provided by the negotiation approaches. Here, features and resources are represented by agents able to communicate with each other to negotiate on their goals. Successful negotiation means that an interaction has been resolved (or that none existed).

In an early paper [Vel93], Velthuijsen evaluates a number of distributed artificial intelligence techniques (DAI) to help resolve the feature interaction problem. Several approaches have since been developed using DAI techniques [BAE+98, AKGM00]. Griffeth and Velthuijsen also use negotiating agents to detect and resolve interactions [GV94]. A resolution is a goal acceptable by all parties and is achieved by exchanging proposal and counter-proposals amongst the agents. Different methods for negotiation have been envisioned: direct (agents negotiate directly without a negotiator), indirect (a dedicated negotiator controls the negotiation and can propose solutions based on past experience) and arbitrated (an arbitrator takes the scripts of the agents and has sole responsibility to find a solution). Griffeth and Velthuijsen concentrate on indirect negotiation. The negotiation approach has been implemented in the "Touring Machine" platform [ABB+93], though no conclusive report about success (or otherwise) is provided.

Rather than using direct negotiation, Buhr et al. [BAE+98] make use of a blackboard. Features are represented by agents which exchange information by writing to a public

data space (the blackboard). Other agents can change the information written to the blackboard and a common goal can be negotiated. The success of the technique is reflected in its use in Mitel's MediaPath product. Amer et al. [AKGM00] also use the blackboard technique, but extend their agents to make use of *fuzzy policies*. Agents set truth-values (0 to 100) to express the desirability of certain goals. These values are then adapted as the call progresses, depending on the values of other agents. In the case of conflict, an event with the highest truth-value is executed.

Negotiation approaches are capable of handling interactions between features provided by several vendors, as long as a consistent mechanism for negotiation exists. However, negotiation requires communication between agents or features. The impact is that in general significant architectural changes are required. However, in the context of legacy systems, architectural changes are not an option. Hence the approaches, despite looking promising, fail in this context.

Note that those disadvantages become less relevant when new emerging architectures with APIs such as JAIN (Java API's for Advanced Networks) [JAI] or PARLAY [Par] are considered. Richer protocols (e.g. SIP [HSSR99] or H323 [ITU00]) facilitate for the required inter-agent communication.

A more complete, though slightly outdated, summary of the feature interaction research field is provided by Keck and Kuehn [KK98]. A more recent review by Calder et al. [CKMR01] includes a forecast for the field (we will return to this in Chapter 8).

Of the presented work, Marples's [MM98] approach seems to be the most flexible, as it can handle legacy systems as well as third party features. Furthermore, it adapts to new architectures. However, the major draw-back is the weak resolution mechanism. Our work extends the resolution mechanism. We will now present Marples's approach in more detail.

2.3 Review of the Transactional Approach

The motivation for the transactional approach [Mar00] is that interactions between features developed by independent parties in conjunction with legacy systems should be detected and resolved. The approach is required to work with more than two interacting features.

The requirements are that we have no prior knowledge of the features, that the features must remain unchanged, that the solution can be incorporated within a legacy system and that it can be employed in a runtime system. This last requirement implies that a mechanism to deal with unexpected behaviour is required. Marples allows features to "fail", as long as they do safely – i.e. they do not affect the rest of the network and do not confuse the user. There must be a way of recovering from unexpected behaviour (e.g. the user going onhook) and the network operator must be able to prevent this from taking place again in the future.

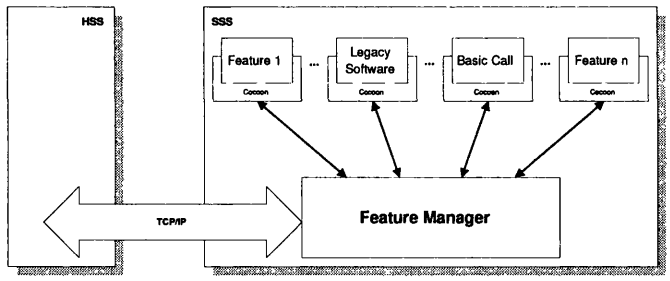

Fig. 2.1: Software Architecture of DESK

Marples does not provide an automatic resolution mechanism. Instead his method detects an interaction and then determines whether this interaction was seen previously (e.g. from information held in a database). If the interaction has been seen before, the earlier resolution is retrieved (e.g. looked up in a database) and applied; otherwise the possible solutions are presented to the network operator.

The experimental results are based on the DESK testbed. DESK was developed by Marples et al. [MMS95, MTMS95] to experiment with feature interaction techniques. The initial target was interaction detection, which explains the crude resolution techniques.

DESK consists of two major subcomponents, the hardware sub system (HSS) and the software sub system (SSS), as shown in Fig. 2.1. The HSS represents the terminal devices, the SSS the switch. A detailed description can be found in [CMRT02], however much of the detail provided therein is not relevant here. What is important is that DESK assumes a single point of call control. That is, there is a central point at which one can observe all messages that are received from or sent to any user in the system. This is where the feature manager is located.

The features and legacy software are encapsulated in a transactional *cocoon*. It is this idea that allows features to remain unchanged. The feature manager communicates directly with the cocoons, using essentially two types of messages: *control messages* that steer the transactional process and *feature messages* that represent actual telecommunications events.

At runtime, the feature manager passes incoming messages from the switching hardware to all the cocoons – for simplicity we assume that this happens in parallel.

The cocoon passes feature messages on to the encapsulated feature. When a message is processed by a feature two possible behaviours can occur: the message triggers a response (one or more messages) or it does not. All triggered responses are sent back to the feature manager, concluding with a *transaction finished* message. The latter is also sent by features not responding with a proper message[2]. Before sending a feature message the feature manager sends a *start_transaction* message which forces

[2] Any message apart from the *transaction finished* message is a proper message.

the cocoon to create a copy of the feature (thus enabling it to revert to the feature's previous state).

The responses are collected by the feature manager, added to a list of responses, and, once all responses are collected, they are evaluated.

There are three possible outcomes:

1. the feature manager did not receive any proper messages,
2. the feature manager received exactly one proper message,
3. the feature manager received multiple proper messages.

The first case occurs when no feature produced an observable response to the trigger event, the call will progress as if no feature manager was in the system. It is the latter two cases which are interesting. The possibility that more than one feature might reply and the consequent potential for an interaction is indeed the motivation for the feature manager. Rather than discuss the resolution of an interaction at this point, we explore the *space* of possible resolutions in the following way.

The feature manager stores the current state and initiates copies of the feature processes. The current state includes the list of events and other local information; it describes a *snapshot of the system from the viewpoint of the feature manager.* Assuming that at least one message has been received, the first message is fed to the copies of the features (again after issuing the *start_transaction* message). The responses are then gathered and processed. Part of this processing may involve further message exchanges with the (copies of the) features.

This process terminates when there are no further responses. At this point, we have a sequence of messages and responses, which we consider as a branch in a *behaviour tree.* To construct the rest of the tree, a rollback to an earlier state is initiated by the feature manager (a *rollback* message is send to the cocoons), the next event is farmed out (Marples's term), a new branch is generated, and so on. Once all events have been processed a full *behaviour tree* has been constructed.

When a single trigger led to more than one response, Marples refers to a Shared Trigger Interaction (STI), if a fed back response triggers another feature he refers to a Sequential Action Interaction (SAI). Indeed, he considers a third group of interactions, Missed Trigger Interactions (MTI) which occur when a feature does not respond because another feature concealed the trigger point. His approach addresses the first two groups [Mar00].

It remains to select the most promising path in that tree, which constitutes the resolution. As described earlier, this is done manually by an operator choosing a branch from the behaviour tree.

When the path is selected, it is committed by sending out the appropriate messages to the hardware subsystem, ensuring that features are in the right states.

The approach depends on certain assumptions about the system, namely that it is possible to create copies of all the associated processes. This might not always be possible, particularly in a legacy system. However, one might emulate process copies if the system provides the functionality to reset a process to a certain state. In this case, in order to emulate multiple copies, stacks of all messages are maintained, the control process is reset and messages are replayed from the stack to obtain a process in the desired state. This is very much like failure recovery in database systems employing a rollback-commit strategy. For further details of the construction of the behaviour trees, see [CMM99].

The major disadvantage of this approach is that there is no automatic resolution mechanism. Resolution involves an operator selecting branches from a tree (and maybe looking up previous resolutions in a database).

A further disadvantage is the restriction to single point of call control, which reflects PBXs (Private Branch Exchanges) accurately, but does not exist in PSTNs (Public Switched Telephone Networks).

Overall, Marples's approach provides very promising technology, assuming that these two weaknesses can be remedied. This is the goal of this dissertation.

2.4 Summary

We have shown that the community investigating the feature interaction problem is very active. Many different approaches have been used to find solutions to the problem.

A special emphasis was placed on the classification of the problem in the benchmark paper [CGL+94], a widely accepted *classification of the problem* of feature interactions, whereby some shortcomings have been highlighted and a new taxonomy was defined. We have not discussed the literature that concentrates on building a *classification of solution approaches* as this seems of little benefit for the current work.

Solution approaches from three different viewpoints – service engineering, formal reasoning and runtime – have been introduced and their advantages and disadvantages discussed. We have then presented a particular runtime approach, namely the transactional approach of Marples [Mar00] – which forms the foundation for this dissertation.

Chapter 3

Running Example

3.1 Introduction

In this dissertation we use a number of features as a benchmark. In this chapter we introduce the benchmark set of features, summarise the meaning of control messages and describe the scenarios which will be considered.

3.2 Features

Large numbers of features can be imagined, and indeed have been developed by providers and developers of telecommunications equipment. For example, the advertisement for Lucent's PathStar switch announces more than 80 features [Luc]. However, detailed specifications are rarely publicly available. For this reason we have to draw on published resources containing a small number of well understood features. An obvious choice is provided by the feature sets from either of the two FIW feature interaction detection contests: the first in 1998 [GBGO98] and the second in 2000 [KMMR00].

Considering the 59 PathStar features for which names (but no details) are provided, we can identify the following classes of features (numbers in brackets show how many features are in the respective class):

- POTS basic call(1)
- Screening features, such as anonymous call rejection (6)
- Group ringing features, such as hunting and group ringing (8)
- Call waiting features (3)
- Call forwarding features (4)
- PC telephony features (4)
- Billing and call logging features, such as call record generation (3)
- Dialling features, such as hotline and direct dialling (16)
- Calling identity delivery and tracing (9)

- Three way calling (1)
- Automatic Callback (1)
- Ringing features, such as distinctive ringing (2)

We have decided to use the features of the second contest, because they are described by finite state machine description techniques. In contrast, the CHISEL [AGG+98, Tur00] notation used in the first contest is relatively unknown, especially outside the telecommunications domain.

The contest provides a set of 12 features in addition to the basic call (which we also treat as a feature). A further feature, *Calling Number Display*, is also introduced. Some features are composed of two parts: a subscriber part and a part located at every user to provide additional behaviour. An example is *Terminating Call Screening* where the subscriber screens certain incoming calls and the information that a caller is screened needs to be relayed back to the subscriber. In order for the caller to evaluate the response he has a general enhancement to basic call, i.e. the part located at every user.

The contest features cover all of the classes identified for the PathStar switch, but ringing, PC telephony and dialling features. As dialling and ringing form part of some other features they are partly covered. PC telephony features are not relevant in the classic telephony context. Thus, the contest features provide a reasonable coverage of currently available features.

Below we informally describe the features, and discuss some general issues related to modelling the features. The detailed formal descriptions can be found in Appendix A.

Basic Call allows for call setup, teardown and basic connectivity between two users. It is often referred to as POTS (Plain Old Telephone System). Note that it is symmetric, i.e. both sides can terminate a call (not currently the case within the BT network).

Call Forwarding on Busy. All calls to the subscriber's line are redirected to a predetermined number when the subscriber's line is busy. For billing, it is assumed that the subscriber pays the charge for the forwarded call from his location to the location to which the call has been forwarded.

Calling Number Display requests the caller's number for display, assuming that the subscribers telephone has a facility to display the number.

Calling Number Delivery Blocking. Usually the caller's identity is available at the terminating side, for evaluation by the callee if required (e.g. caller number display or terminating call screening). This feature blocks the provision of the callers number at the terminating side.

Call Transfer allows the subscriber to transfer the current call to a third party. That is, while being in a call the subscriber can put the second party on hold and can setup a call to a third party. Once the subscriber goes onhook the second party of the first call and the third party are connected. This is effectively a mid-call call forwarding. The subscriber may transfer a call independent of being caller or callee in the original call.

Call Waiting. The subscriber is notified of an incoming call while he is busy in a conversation and can accept the new call by putting the originating call on hold. The subscriber is then able to toggle between the two calls.

Group Ringing allows for an incoming call to ring at three phones. The phone which goes offhook first is connected to the calling party. The remaining two phones stop ringing.

Reverse Charging is also known as freephone billing, and allows the subscriber to be charged for all calls in which the subscriber is the terminating party.

Ringback when Free. When a call attempt is made to a busy subscriber with this feature subscribed, the caller is informed that he will be called back, as soon as the other person becomes available. Once the subscriber terminates his/her current call a connection to the stored number will be established.

Split Billing allows costs to be shared between the partners in a call. A company might provide local call charge lines to customers as a service, in which case the customer (and originator of a call) pays the local charges and the company the rest.

Teenline. During a pre-set time of day, this feature restricts all outgoing calls from the subscriber's telephone. To place an outgoing call during that time a PIN is required.

Terminating Call Screening. The originators of all incoming calls to the subscriber's telephone are screened against a screening list. If the originator of an incoming call matches an entry in the list, then an announcement is played to the originator and the call is cancelled.

Three Way Calling allows a user already connected to bring a third partner into the call. Any side of the first call (as long as it subscribes to Three Way Calling) can set up a connection to the new party. This is established by putting the current partner on hold, connecting to the third side and then joining both lines. The three way call is terminated with any party going onhook.

Voice Mail. Voice mail works in a similar way to an answering machine, by offering the possibility to leave messages if the called user is not available. The stored messages can be played back by the subscriber.

3.2.1 The Role of States

In this section we discuss the problems introduced by the use of states. We show a number of examples and then generalise the observations.

The detailed description of features in the contest is dependent on state information. For example, *Call Forwarding on Busy* is only triggered if the *Basic Call* is in certain states. A feature is integrated into the system by adding or replacing transitions in the *Basic Call* description. However, since state information is not available due to the nature of legacy and proprietary systems, we need to redesign some of the features slightly.

Changes we make are to ensure that the features are triggered and terminated appropriately. The following examples should illustrate this. Note that state labels in the examples refer to the contest description, appendix A contains the updated diagrams.

Example 3.2.1 *Call Forwarding on Busy* is triggered by an incoming alert (*i_alert*) message in a basic call state where the user is **known to be busy**. However, in our setting we are only aware of basic call being busy when the outgoing busy message (*o_busy*) is sent by basic call and then fed back to the features. Hence *o_busy* is a more suitable trigger.

Example 3.2.2 The recording part of *Voice Mail* is described as starting in basic call state BC9 and **immediately sends a response**. However, in the message based description we require this feature will never be triggered. So we require an initial trigger event. Further, repetitive behaviour of the BC (due to the original replacing of states all potential transitions have to be included) can be removed. In this case, *o_timeout*, which is fed back, can be assumed as best trigger event.

Example 3.2.3 The Everyone module of *Call Transfer* exhibits **a choice of initial behaviours**, depending on whether it is located at the originating or terminating end of a call. In this case it is necessary to include new initial trigger events that determine at which side of the call the feature is located. Similar to the previous example we see that the two given original states are either BC7 or BC11, which can be reached by either *i_connect* or the fed back *o_connect*. Hence we will introduce those events as new triggers leading to a new state. In this new state the feature responds to *i_inform* or *i_notify* in the expected way or can be reset to idle by either *i_onhook* or *i_disconnect* (the events which trigger a transaction from states BC7 and BC11)

Example 3.2.4 The final form of change is illustrated by the *Teenline* feature. This feature might result in a terminating state where **the action is a response being sent** (here BC2). However, BC2 is a state from which a *dialtone* is sent so that the call can progress as expected – i.e. the user has the chance to dial a number. We find that the *dialtone* should be included in the teenline features responses (replacing the tau message).

These four examples illustrate problems occurring when state information is not available. We can generalise from the examples to identify the following measures

which have been applied to the feature descriptions from the contest to produce the descriptions in appendix A:

- replace the original trigger with a more suitable one (example 3.2.1),
- introduce a new trigger for those features with an initial state that only allows for responses to be send (example 3.2.2),
- introduce a new trigger that places a feature in a waiting state and allow it to quit this waiting state when a feature has multiple distinct initial states. (example 3.2.3),
- introduce a new response, when a feature would exit into a basic call state that produces a response (example 3.2.4),
- remove messages that only duplicate behaviour which the basic call process would exhibit anyway.

One could argue that our attempts to ensure that a feature is triggered by an incoming trigger message are superfluous. For example an alarm feature which is triggered by an internal clock appears to exhibit spontaneous behaviour. A closer analysis reveals that this is not true: the alarm feature is triggered by an incoming trigger message, namely when the user sets up the alarm. Clearly, this shows that the incoming trigger exists. As we do not consider time between events our effort is justified.

3.3 Control Messages

We can only observe the behaviour of features by analysing the messages that are passed around the system. Thus an informal understanding of the meaning of messages is required.

All messages consist of two parts: the event and an argument. A "-" denotes a null argument.

There are three main groups of messages. Firstly messages exchanged between the terminal devices and the features (and vice versa) – we refer to them as *terminal messages.* Secondly, messages exchanged between features associated with different terminal devices, referred to as *feature messages.* Thirdly, messages with billing information (*billing messages*). Each of these types is discussed in more detail below, the list is exhaustive for the given features though new features might introduce additional messages.

3.3.1 Terminal Messages

A terminal is used as a device which provides the interface between the user and the network model. We cannot distinguish between actions from the user and the device. We distinguish between messages originating from the terminal and those initiated by the features.

User initiated messages:

(offhook, -) the terminal has gone offhook.

(dial, number) the terminal dialled the number *number*.

(onhook, -) the terminal has gone onhook.

(flash, -) the terminal flashes, that is goes onhook briefly and then offhook again. Usually this behaviour is created by pressing the flash button on a phone.

Feature initiated messages:

(dial_tone, -) a dialtone has been initiated.

(ringtone, -) a ringtone has been initiated.

(busytone, -) a busytone has been initiated.

(timeout_tone, -) a timeout_tone has been initiated.

(disconnect_tone, -) a disconnect_tone has been initiated.

(connect, -) a connection has been established.

(stop_alert, -) the terminal stops ringing.

(alert, -) the terminal starts ringing.

(announce, message) an announcement message is played. For instance, a notification that the called party is not available but a message can be left.

(display, information) some information is displayed, e.g. the callers identity. It is assumed that the terminal device has a suitable display facility.

(cwtone, -) a call waiting tone has been initiated signalling that the call is currently on hold or that another call attempt is being made (depending on the situation of the user).

(store_msg, msg) the message (voice) from the calling party is stored in the mailbox.

(store_read, msg) the message (voice) from the mailbox is transmitted to the terminal.

(store_clear, -) the message stored in the mailbox is removed.

Note that tones are stopped when any subsequent message is sent or received.

3.3.2 Feature Messages

Messages commencing with "*o_*" are sent by a feature, those starting with "*i_*" are received by a feature. All of these messages come in pairs due to the fact that they are the same message, just with a different direction from the viewpoint of a particular feature. Note that the conversion from "*o_*"-messages into "*i_*"-messages is performed within the switch and is not of concern here. We will only describe the meaning of outgoing messages, i.e. those starting with "*o_*". Clearly the recipient of the messages is the (potential) partner in the call.

(o_alert, -) notifies that a call attempt is being made.

(o_stopalert, -) notifies that a call attempt is being dropped.

(o_disconnect, -) notifies that a call (that was connected) has been terminated.

(o_connect, -) notifies that a call has been answered, i.e. the called party has gone offhook.

(o_timeout, -) notifies that the system has timed out a connection attempt due to the call not being answered, within the system internal timebounds.

(o_busy, -) informs about busy status, in this case the called party is busy.

(o_free, -) informs about busy status, in this case the called party is free.

(o_inform, information) communicates information. It is used here for features that expect an announcement or other notification to be made at the other side of the call.

(o_msg, msg) prompts the voicemail feature to transmit messages to be stored.

(o_notify, Z) notifies that a call is being forwarded to Z.

(o_request, query) requests information, for example the caller identification.

3.3.3 Billing Messages

Billing events are sent from the features to the billing system – we are not concerned with details of the billing system here. The global time is passed as a parameter so that the duration of some activity can be measured and a corresponding charge levied. Note that we have abstracted the messages slightly, so that they no longer contain information about the users involved. Clearly, this information would be needed to create correct bills, but as we are only concerned with interaction this is not required here.

(billing_start, time) Start charging for a call.

(billing_stop, time) The counterpart to the *billing_start* event. Upon receipt of this event the duration of a call can be calculated and the user be charged with the correct amount.

(billing_forwarded, Z) The billing system is notified that a call has been forwarded to user Z. Hence the forwarding can be taken into account for billing.

(billing_reverse, -) The billing system is notified that for the next event *(billing_start, time)* the terminating side of the call is charged.

(billing_split, factor) The billing system is notified that for the next event *(billing_start, time)* both users are charged according to the factor. For instance, the subscribing user will pay 30% of the charge if the factor is 70. In addition the factor can be used to code policies like the caller only pays a local call charge and the subscriber the remainder etc.

(billing_onhook, time) The billing system is notified that a terminal is going onhook.

(billing_offhook, time) The billing system is notified that a terminal is going offhook.

3.4 Scenarios

Scenarios describe configurations of the telephone system to varying levels of detail. They are usually created in a rather pragmatic way, whereby the experience and intuition of the designer has a major impact on the quality. Scenarios are required for evaluation of features as well as for feature interaction detection and resolution methods. Scenarios are valuable for testing, and are indeed used in current testing practice.

A typical scenario describes the system configuration and the state of the users. Consider the following example:

Example 3.4.1 Assume four users: A, B, C and D. B subscribes to both Call Forwarding Busy (initialised to forward calls to C) and Call Waiting. B is in conversation with D. A rings B.

It is very easy to invent scenarios, but it is not possible to ensure that all possibilities are covered. It seems possible to use combinatorial techniques to list all cases. However, this requires a framework that defines how many users must be considered in order to ensure that all interactions between two or more features are detected. Such a framework does not exist.

Current interaction detection approaches mostly consider 2-way interactions, that is where 2 features are subscribed to by the users in the system. In addition, some work on 3-way interaction has been done and there has been much discussion concerning the desirability of 3-way interaction analysis. It is basically argued that "true" 3-way interactions [TM00], defined to be those where no interaction exists between any pair of the concerned features, are extremely rare.

A runtime approach must be able to handle any number of simultaneously subscribed features, hence we propose analysing n-way interactions. This is motivated by the fact that large numbers of features are available to users.

We will revisit the idea of scenarios when evaluating our approach.

3.5 Summary

We have identified a set of features, provided their informal specifications and discussed the concept of scenarios.

Chapter 4

A Hybrid Approach to FI

4.1 Introduction

As discussed earlier, feature interaction techniques can be classified as offline or online, each with their own strengths and weaknesses. In this thesis the aim is to combine the strengths of both and thus reduce the weaknesses – the result is a hybrid approach.

This chapter gives an overview of the envisioned hybrid approach, and discusses the advantages of such an approach. A suitable online detection method is available from the work of Marples [Mar00]. However, as shown earlier, the method's resolution approach is weak. Here we show how the hybrid approach allows for further development of the detection technique and how it supports building a stronger resolution method, thus resulting in a comprehensive solution for dealing with feature interactions.

4.2 A Hybrid Approach

We expect our feature manager not only to detect interactions between new features and the legacy system, but to resolve detected interactions in a satisfactory way. Sometimes the resolution will involve the suppression of a number of features, raising the question: which ones? At other times it may be possible to interleave features or to run them in a particular order. In any of these cases the feature manager must be able to make those decisions.

For detection, we follow the transactional approach proposed by Marples [Mar00] in which the legacy system and the new features are treated as black boxes embedded in transactional cocoons. The cocoons permit rollback and commit facilities. This allows one to experiment at runtime with different sequences of possible inputs and thereby to choose the best resolution.

Building on this detection mechanism, we aim to develop resolution methods. The information required in order to choose a "best" resolution is derived from an offline analysis of feature behaviour. This information is in the form of general rules (referred to as *resolution rules*) and thus is independent of the features actually deployed. The main contribution of this thesis is to show that this is indeed possible and to show how it can be achieved.

The rules may be gathered from an analysis of the behaviour of a formal model of the system.

The formal model is more than just an abstraction, or specification, of intended or required behaviour. It is an integral part of the incremental process of the development and refinement of those intentions or requirements. That is, it is part of a adaptive, experimental process, as advocated by Calder [Cal98] and Calder and Reiff [CR00]. Figure 4.1 illustrates this process.

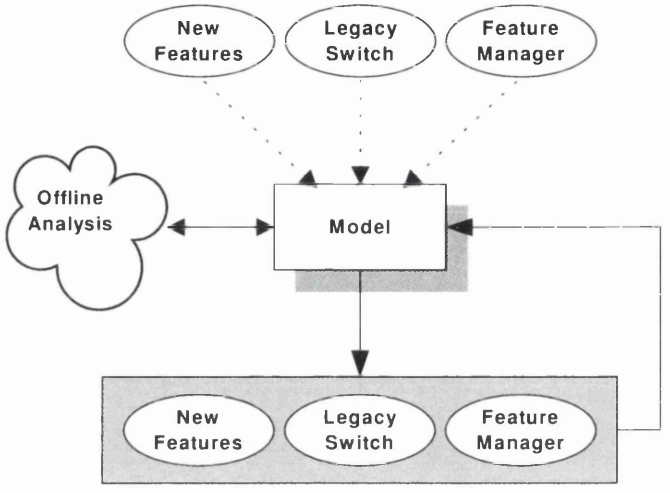

Fig. 4.1: Adaptive Development Process

The initial step in the process is to develop an (initial) formal model of the online system, i.e. the legacy switch, a relatively uninformed feature manager, and the features. Dotted lines denote this step. This model provides us with a platform from which to experiment and reason about observable behaviour of the legacy system and the new features.

The feature manager in this initial stage is concerned with identifying all possible solutions, i.e. constructing the solution space, as shown in Fig. 4.2.

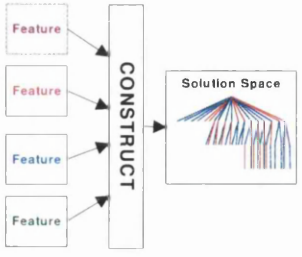

Fig. 4.2: The Initial Behaviour of the Feature Manager

In the model the input to the feature manager is formed by example features. Implementation details of these example features are assumed to be known to allow for

analysis. The feature manager constructs the solution space by probing the features, thus enabling that (in an operational system) the knowledge of the feature behaviour is not required.

Properties of this system enable the identification of resolution rules such as "event x should never be followed by event y, otherwise inconsistent behaviour is given to the user" or "events x and y should never be offered simultaneously" (e.g. a spoken announcement and a busy tone).

In the next step, the derived rules are used to guide the feature manager. That is the uninformed feature manager will be replaced with a better informed version incorporating algorithms to resolve interactions. This alters the system behaviour, so we can derive new rules, further enhance the feature manager, observe more behaviour, and so on.

4.3 Detection and Resolution Process

The detection and resolution process is represented by Fig. 4.3. The construction of the solution space is identical to that used by the uninformed feature manager.

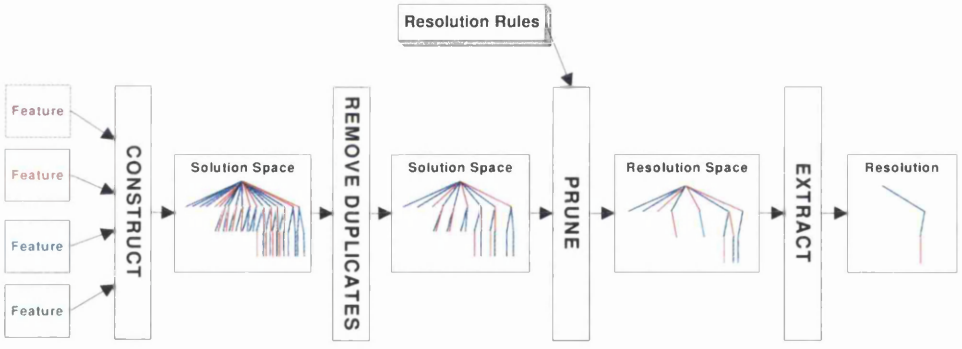

Fig. 4.3: Detection and Resolution Process

During the offline analysis we identified rules that are applied to reduce the solution space to a set of valid solutions. Finally, the best (valid) solution is selected. The rules themselves can be refined subsequently by a better understanding gained from applying them, leading to the iterative nature of the adaptive development process.

We stress that at runtime the implementation details of the features are unknown. The feature manager's explorative method allows the identification of all possible solutions.

4.4 Feature Interactions

Feature interaction simply refers to the fact that there is a "point of contact" between two or more features, not discriminating between desired and undesired interactions.

In general it is difficult to decide whether or not an interaction is desired, as factors like user expectations play a major role in this classification. Our approach for detection is based on these contact points, but for resolution it will be necessary to distinguish between desired and undesired effects.

We identify two kinds of interactions, which we define as follows:

Definition 4.4.1 (Technical Interaction) A technical interaction occurs when several features triggered by the same event or features triggered by an earlier response request that the call is continued in distinct, non-unifiable ways. That is, there is no system state which satisfies the behaviour of all requests.

Definition 4.4.2 (User Intention Violation) A user intention violation occurs when a user observes unexpected behaviour in a call that progresses (i.e. where no technical interaction has occurred).

Clearly both can be seen as an interaction, as there is a point of contact between the features. (We note that Marples does not distinguish between these two types of interaction.) Let us consider some examples to illustrate the difference:

Example 4.4.1 (CT-RC) Assume user B subscribes to *Call Forwarding* which forwards all incoming calls to user C. C subscribes to *Do Not Disturb*, which plays a polite message informing that C is currently unavailable. A calls B and gets forwarded to C. A now hears C's message. Clearly being connected to C is not what A expected, hence we could classify this as a **user intention violation**. This is not a technical interaction, as both features behaved correctly and more importantly the system is not placed in unexpected states (thus remaining stable and allowing the call to progress).

Example 4.4.2 (CFB-CW) User A subscribes to both *Call Forwarding Busy* and *Call Waiting*. A is talking to B and receives a call from C. This causes a **technical interaction**: a forwarding attempt and the announcement of a call waiting tone. Each of these responses is invalidated by the other, there is no meaningful state of the system satisfying both behaviours simultaneously. Namely, if the call is forwarded, the call is not in a waiting position and vice versa.

Clearly in the context of multiple features, both kinds of interaction might occur simultaneously, for example upon receiving trigger t features f_1 and f_2 produce a technical interaction and features f_1 and f_3 lead to a user intention violation. To resolve technical interactions, all but one feature involved in the technical interaction must be disabled, in the example f_3 and either f_1 or f_2 can proceed.

We will concentrate on detecting technical interactions. Note that in new systems it would be possible to express user intentions in form of policies, which however will shift the problem from feature to policy interactions.

In sections 2.2.4 and 2.3 we have outlined other classifications for interactions, namely Shared Trigger Interactions (STI), Sequential Actions Interactions (SAI) and Missed

Trigger Interactions (MTI)[Mar00] and the more classic classes (defined in [CGL+94]): MUSC, SUSC, MUMC and SUMC.

Before showing how these classifications relate to our distinction of user intention violation and technical interactions, we need to discuss two issues: call control points and a notion of *stable state.*

Points of call control. A call can be controlled either from a single point (as in a PBXs) or from multiple points (as in a PSTN). The number of call control points impacts on the availability of data and the required messages. As an example consider a call attempt to a busy user. In a single point of call control (SPCC) setting, the switching software simply checks the status of the called user in a table and supplies the caller with ether a busytone or a ringing tone. In a multiple point of call control (MPCC) setting the caller's switching software generates an outgoing call attempt and awaits a response from the callee's switch containing information regarding the busy status of the callee. DESK uses SPCC, our running example employs MPCC.

The impact of call control points on interaction detection and resolution is significant: in a SPCC setting features from both involved parties can be queried, that is all data available to them can be accessed and resolution mechanisms can influence features at both ends of the call. In MPCC settings this is not possible, the only data available is that belonging to the local user and that transmitted from the remote end via messages. Additional information might be queried, resulting in a further message exchange. Furthermore, a resolution involving state changes or termination of remote end features is in current telephone systems not possible – only local features can be influenced.

Stable states. For SPCC a stable state is one where a user input can be received – i.e. the last user event has been processed and all internal messages have been consumed. In Marples's [Mar00] work this is described as *a state with no events or responses pending.* In MPCC a stable state is one where either a user or a remote switch event can be received – again all previous events have been dealt with. The impact of this seemingly small change can best be shown with an example: consider two users with call forwarding features (for simplicity assume unconditional forwarding). A forwards to B and B to A. When A receives a call it is forwarded to B. The next event will be a forward from B to A. In a SPCC setting this occurs *before* the next stable state is reached, in MPCC *after* the next stable state was reached.

Our detection and resolution algorithm does not span across stable states (we cannot revert a decision that has been made after it is committed to the network). Thus, we are able to detect the forwarding loop with our approach in a SPCC setting but not in a MPCC setting. We will consider this in more detail when evaluating our approach in section 7.6.

The given classifications of interactions are orthogonal, they all divide the same space of possible interactions into different partitions. Cameron et al. [CGL+94] defined the classes for IN networks (which assume MPCC), Marples for SPCC. However, Marples classes can be mapped onto MPCC settings.

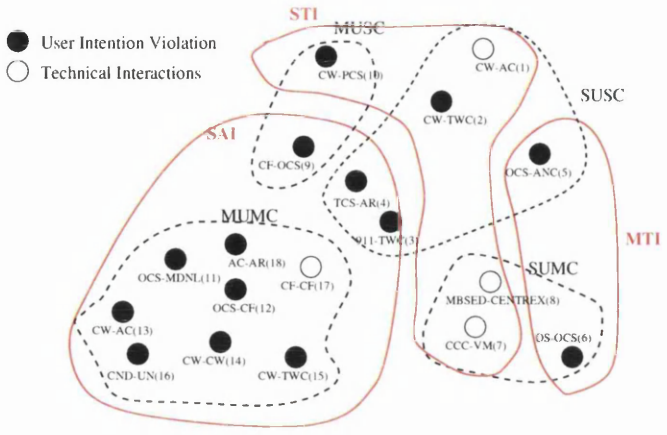

Fig. 4.4: Detection and Resolution Process

Analysing the feature interactions described in Cameron et al. [CGL+94] we obtain overlaps as shown in Fig. 4.4. Notably no class completely encapsulates any of the others – with the exception that all MUSC interactions are user intention violations. All three classifications are suitable for classifying feature interactions independent of the number of call control points and the specified notion of stable states.

4.4.1 Aside: The Feature Manager and Call Control in the IN standard

We have discussed the idea of single versus multiple points of call control and also that of a feature manager. Features and their handling have been envisioned in the current standard for intelligent networks (IN) [ITU93b], where the above ideas are also introduced. IN supports both single point of call control and multiple points of call control – the former is relevant for Private Branch Exchanges (PBXs), the latter for the Public Switched Telephone Network (PSTN). Multiple points of call control can be seen to be more important, as a PBX is usually a proprietary product and hence an international standard describing its working is less relevant. Further, multiple points of call control are technically more challenging, as one side only has a restricted or abstract view of the other side in the call. Calls are typically made up of segments from one switch to another where each switch has independent call control and no direct control over actions at another switch (see Fig. 4.5). Obviously some protocol is required for correct inter-operability of two switches on shared segments.

A close look at the SSF/CCF (service switching function/call control function) part reveals a complex internal structure (Fig. 4.6). The basic call manager (BCM) handles basic call setup and connectivity, but also recognises IN trigger events and passes these to the SSF. The BCM has associated resources and data and also communicates with the Bearer Control (which takes care of the interaction with the medium).

More interestingly, IN triggers passed to the SSF are received by a feature interaction

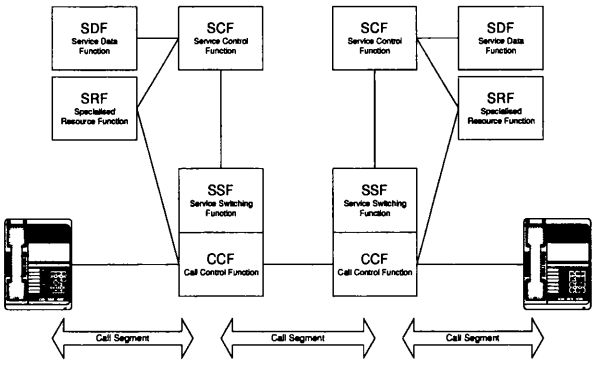

Fig. 4.5: Overview: end-to-end call in IN networks

manager (FIM). The FIM communicates via the IN service switching manager (IN-SSM) with the service logic (also referred to as service control function (SCF)). The SCF is basically what we refer to as features. Similar to the BCM, the IN-SSM has access to private resources and data. The IN-SSM provides the features with an abstract view of the call as well as access to its call control and its own resources.

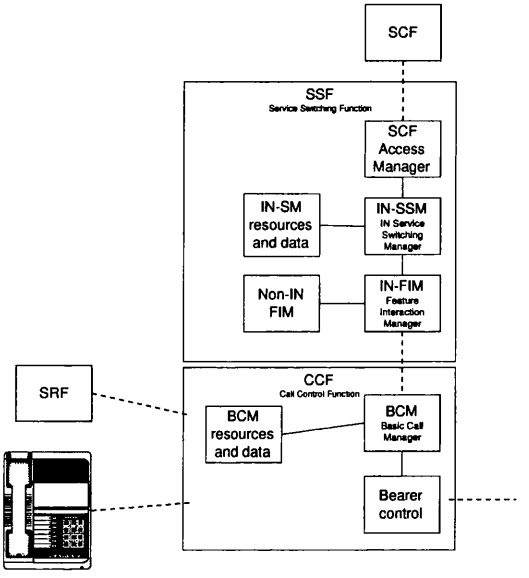

Fig. 4.6: Call Control and Service Switching Functions in IN

The feature interaction manager/call manager (FIM/CM) is "the entity that provides mechanisms to support multiple concurrent instances of IN service logic instances and non-IN service logic instances on a single call. In particular the FIM/CM can prevent multiple instances of IN and non-IN service logic instances from being invoked. The ability of the FIM/CM to arbitrate between multiple instances of IN and non-IN service logic instances is for further study. The FIM/CM integrates these interaction

mechanisms with the BCM and IN-FM[1] to provide the SSF[2] with a unified view of call/service processing internal to the SSF for a single call." [ITU93b, p6]

Let us analyse this description in more detail. The FIM can coordinate events reporting to the IN-SSM (thus influence the abstract view), impact on BCM (by coordinating suspension or resumption of the basic call) and provide:

- a service logic instance selection mechanism to determine which service logic to invoke or block (a targeting of trigger events to individual features),
- mechanisms to support simple restricted service logic interaction between simultaneous active features on the same call segment and
- mechanisms to prohibit simultaneous active features using static or dynamic mechanisms.

Static mechanisms are basically service management functions (via provisioning), normally based on resolution tables. The tables show potential problems between features. Dynamic mechanisms "may involve more complex capabilities" [ITU93b, p34]. Proposed dynamic methods are priorities and precedences of features or exclusion of features (i.e. prohibiting new ones while others are still active).

In conclusion, our approach can integrate the basic ideas of IN (the standardised architecture provides similar, but cruder, concepts). We extend the defined (IN) feature manager by providing the afore-mentioned "more complex capabilities", thus giving the feature manager a stronger resolution mechanism. Further, our approach removes the need for the feature manager to know details about the features, which the approach in the IN standard [ITU93b] clearly requires. We also address the issue of arbitration between multiple features which was left for "further study" in the IN standard.

4.5 Summary

In this chapter we have described the concept of a hybrid approach. We have discussed a classification of interactions and call control issues. Our approach improves on Marples detection work, in that we consider more realistic call control based on the IN approach.

[1] We assume a typing error in the document, the component referred to is the IN-SSM

[2] We assume a typing error in the document, the component referred to is the SCF

Chapter 5

Potential Resolutions

5.1 Introduction

In this chapter we will consider several issues concerned with the question: *What are potential resolutions when an interaction has been detected?* Recall that detected means that more than one feature responded to a trigger or that at least one feature responded to a fed back response. This chapter makes the idea of detected interaction more precise.

The specification of a solution space, a set of all possible solutions, leads to an initial approach of constructing the same. This provides the precise understanding of detected interactions and solutions, but is somewhat restricted in comparison to realistic features. Thus for further analysis a better model needs to be found. Process algebra seems a very good candidate notation, so we discuss why have chosen Haskell instead. Finally, details of the Haskell implementation are discussed.

5.2 The Solution Space

Marples's idea of exploring whether several interacting features might be allowed to proceed in a certain order, inspired our goal to allow as many interacting features to proceed as possible. To this extent a sound understanding of the possible solutions is required. Let us make precise the terms solution, solutions space and resolution.

For a given set of features, a **solution** is a trace of one or more of those features running concurrently. That is, it is an interleaving of messages generated by a subset of the features.

For a given set of features, the **solution space** is the set of all traces, for all subsets of the features.

A very simple solution space (for 3 features, each of which is executed "atomically", i.e. there is no interleaving here) is illustrated in Fig. 5.1. It should be noted that the solution space might contain many traces that lead to a violation of required properties (i.e. there might be traces that represent incorrect behaviour).

For a given set of features, a **resolution** is a trace in the solution space that does not violate any specified properties.

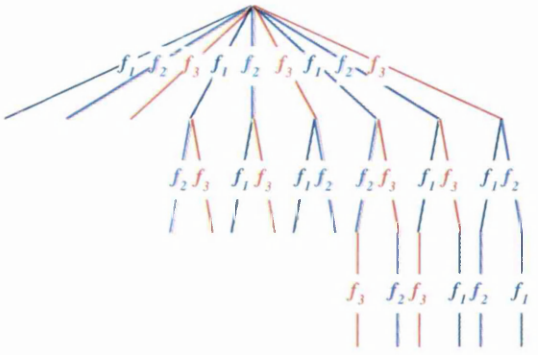

Fig. 5.1: Simple Solution Space

The solution space depends on the granularity of the interleaving. A coarse grained interleaving would allow features to be run in any order but does not allow messages of features to be interleaved. In this case the solution tree grows exponentially with the number of interacting features. A fine grained interleaving would also allow individual messages to be interleaved thus resulting in a solution space growing exponentially in both the number of features and the length of their responses. We adopt the latter, which clearly includes the former.

5.3 Specification of the Solution Space

In this section we develop a specification of the solution space: we consider messages as basic building blocks, then features. The solution space is the result of a feature manager function applied to a number of features.

5.3.1 Messages

A message consists of an event (a type and possibly a value) and an associated I/O aspect. Examples of event types are dial or alert; the value depends on the type (e.g. *dial* has the dialled number as value, *dial-tone* does not require a value, and *announce* has an event value indicating the announced message). The I/O aspect indicates whether the message is input to or output from a feature.

Convention: input messages are preceded by +, output messages are preceded by −.

We write a message m as $(\pm, \textit{event type}, \textit{event value})$ or, when we are only concerned about the I/O aspect, as $+m$ or $-m$. The null value is denoted by "-" and, by abuse of notation, τ is also a special null message (τ is an output message).

Example 5.3.1 The following are valid messages: $(+, \textit{dial}, n)$, $(-, \textit{announce}$, "screened") and $(-, \textit{i\_alert}, \text{-})$.

We require relations between messages. Two messages a and b are said to be

equal when their event types, values and their I/O aspects are equal, written $a = b$.

duals when their event types and values are equal but their I/O aspects are not equal, written $a \doteq b$.

Example 5.3.2

- $(+, i\_alert, \text{-}) = (+, i\_alert, \text{-})$
- $(+, i\_alert, \text{-}) \doteq (-, i\_alert, \text{-})$

5.3.2 Features

Definition 5.3.1 (Alphabet) An alphabet is a set of messages partitioned into 2 non-empty sets: input messages and output messages.

Definition 5.3.2 (Trace) A trace over an alphabet is a non-empty finite sequence of elements of the alphabet starting with an input message. $First(t)$ is the first element in a trace t.

Convention: A trace is obtained by juxtapositioning elements of the alphabet.

Example 5.3.3 Consider the alphabet:

$\alpha = \{(+, i\_alert, \text{-}),\ (-, announce, \text{“screened”}),\ (-, o\_request, \text{“callerid”}),$
$(+, i\_inform, \text{“id”}),\ \tau\}.$

Some of the possible traces over α are:

$(+, i\_alert, \text{-})(-, announce, \text{“screened”})$ and
$(+, i\_inform, \text{“id”})(-, announce, \text{“screened”})(-, announce, \text{“screened”})$

The *Terminating Call Screening* feature is defined by the trace set:

$F_{TCS} = \{(+, i\_alert, \text{-})(-, o\_request, \text{“callerid”})(+, i\_inform, \text{“id”})(-, announce, \text{“screened”}),$
$(+, i\_alert, \text{-})(-, o\_request, \text{“callerid”})(+, i\_inform, \text{“id”})\,\tau\}.$

Definition 5.3.3 (Feature) A feature is a set of traces over an alphabet. We assume the alphabet is minimal (i.e. each element of the alphabet occurs in at least one trace).

Convention: feature F_i has alphabet α_i.

Example 5.3.4 This example shows the trace sets for some features. The label at the right column will be used later to reference the respective trace.

Split Billing:

$F_{SB} = \{(+, dial, number)(-, billing\_split, \text{factor}),$ $t_{(1)}$

$(+, dial, number)\,\tau\}$ $t_{(2)}$

Reverse Charging:

$F_{RC} = \{(+, dial, number)(-, billing\_reverse, \text{-}),$ $t_{(3)}$

$(+, dial, number)\,\tau\}$ $t_{(4)}$

Calling Number Display:

$F_{CND} = \{(+, i\_alert, \text{-})(-, o\_request, \text{“id”})(+, i\_inform, \text{“id”})(-, display, \text{“id”}),$ $t_{(5)}$

$(+, i\_alert, \text{-})(-, o\_request, \text{“id”})(+, i\_alert, \text{-})(-, o\_busy, \text{-})$

$(+, i\_inform, \text{“id”})(-, display, \text{“id”})\}.$ $t_{(6)}$

Call Forwarding on Busy:

$F_{CFB} = \{(+, o\_busy, \text{-})(-, o\_alert, \text{-})(-, billing\_forwarded, \text{D})(-, o\_notify, \text{D})\}$ $t_{(7)}$

5.3.3 Feature Interaction

Now we can formalise the concept of an interaction:

Definition 5.3.4 (Feature Interaction) We say that n features $F_1 ... F_n$ with respective alphabets $\alpha_1 ... \alpha_n$ interact iff

$$\exists i, j : ((1 \leq i, j \leq n) \wedge (i \neq j)) \wedge (\exists + a \in \alpha_i \wedge (\exists b \in \alpha_j : a \doteq b))$$

Features interact if they have common input messages in their respective alphabets, or one alphabet contains an input message that is the inverse to an output message in another alphabet. Note that we do not consider the case where a nd b are both output messages. This is because output messages will only occur simultaneously if they are triggered by the same event. Thus a corresponding pair of matching input messages would exist. This definition clearly reflects the previously discussed concept of feature interactions being caused by points of contact between features (section 4.4).

5.3.4 The Feature Manager

The purpose of the feature manager is to detect and resolve interactions. Hence the feature manager has to distinguish between features which do and do not interact in the sense of Definition 5.3.4.

The feature manager is defined as follows:

Let $F_1, ..., F_n$ be features and let e be an input message, where $e \in \alpha_1 \cup ... \cup \alpha_n$.

Definition 5.3.5 (Feature Manager) The feature manager is a function, defined as:

$$FM(e, F_1, ..., F_n) = \begin{cases} Extract(Prune(Construct(F_1, ..., F_n))), e) \\ \qquad \text{if } F_1, ..., F_n \text{ interact} \\ Extract((F_1 \cup ... \cup F_n), e) \\ \qquad \text{otherwise} \end{cases}$$

$Extract(T, e)$ returns a set of traces $T' \subseteq T$ such that $\forall t : (t \in T') \wedge First(t) = e$.

$Prune(T)$ returns a set of traces $T' \subseteq T$, where all the traces in T' represent possible resolutions.

$Construct(F_1, ..., F_n)$ constructs the solution space of features $F_1, ..., F_n$ and will be considered in detail in section 5.4.

It may seem surprising that *Extract* can return more than one t race, after all we are looking for a single resolution. To motivate our definition, consider a simple single feature example: $F = \{+a\,-b\,+c\,-d,\ +a\,-b\,+e\,-f\}$. Both traces are equally good resolutions when the trigger is a, but when the call progresses either c or e will occur as trigger, forcing the other trace to be irrelevant. However, this is not an interaction and we expect the feature manager to return both traces. The decision as to which trace should finally be chosen is dependant on a later trigger event. Therefore, we cannot make the decision at this point and must return a set. Similar scenarios can arise between multiple features, thus leading to the above understanding of *Extract*.

5.4 Construction of the Solution Space

The role of the *Construct* function is to construct the solution space, i.e. the set of all potential solutions. The complexity of the definition reflects the complexity of the task: *Construct* returns a set containing all the individual features traces and all traces representing interleavings of multiple features. The latter are constructed using the function *overlap*, which in turn depends on the concept of overlapping interleavings. We consider the concept of overlapping interleaving and then the functions overlap and construct, showing examples for each.

5.4.1 Overlapping Interleaving

The following definition, illustrated with figure 5.2, makes precise the concept of "overlapping interleaving".

In the following, to increase legibility we substitute $a = a_1...a_n$ for t_1 and $b = b_1...b_m$ for t_2. Let *concat* be a function concatenating sequences. Let i be the first occurrence of the longest prefix of a occurring[1] in b. Let j be the length of that prefix.

[1] a message m occurs in a and b if there are messages a_x and b_y such that $a_x = b_y$.

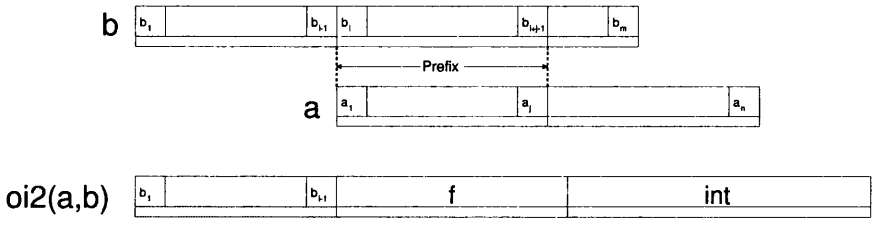

Fig. 5.2: Overlapping Interleavings

Definition 5.4.1 A trace t is an overlapping interleaving of t_1 and t_2, if and only if

$$t = concat(b_1...b_{i-1}, f(i,j,a,b), int(a_{j+1}...a_n, b_{j+i}...b_m))$$

where the function f, which computes the event sequence arising from the prefix, is defined as follows:

$$f(i,1,a,b) = \begin{cases} b_i & \text{if } a_1 = b_i \\ b_i a_1 & \text{if } a_1 \doteq b_i \end{cases}$$

$$f(i,j,a,b) = \begin{cases} concat(f(i,j-1,a,b), a_j) & \text{if } a_j = b_{i+j-1} \\ concat(f(i,j-1,a,b), a_j, b_{i+j-1}) & \text{if } a_j \doteq b_{i+j-1} \text{ and} \\ & a_j \text{ is an output message} \\ concat(f(i,j-1,a,b), b_{i+j-1}, a_j) & \text{if } a_j \doteq b_{i+j-1} \text{ and} \\ & b_{i+j-1} \text{ is an output message} \end{cases}$$

and the function int, which computes an interleaving is defined as follows: $int(s_1, ..., s_n)$, an interleaving of sequences $s_1, ..., s_n$ with length $|s_1|, ..., |s_n|$ respectively, is a new sequence S such that

- $|S| = \sum_{i=1}^{n} |s_i|$,
- $\forall a : (a \in S) \leftrightarrow (a \in s_1 \vee ... \vee a \in s_n)$,
- the relative order of the elements is preserved in S (i.e. if an element a occurs in a sequence before b, then they do so in S).

We are now in a position to present some examples which illustrate the above definition. For readability we do not show the event value.

Example 5.4.1 $oi2(t_{(6)}, t_{(7)})$ = the set of all overlapping interleavings between $t_{(6)}$ and $t_{(7)}$. Note that the *o_busy* output of $t_{(6)}$ is input to $t_{(7)}$. The longest prefix of $t_{(7)}$ in $t_{(6)}$ is the sequence (*o_busy*), occurring at position 4 in $t_{(6)}$. There are several $t \in oi2(t_{(6)}, t_{(7)})$, as we shall show:

$$
\begin{aligned}
t &= concat(+i\_alert - o\_request + i\_alert, f(4,1,t_{(6)},t_{(7)}),\\
&\qquad int(+i\_inform - display, -o\_alert - billing\_forwarded - o\_notify))\\
&= concat(+i\_alert - o\_request + i\_alert - o\_busy + o\_busy,\\
&\qquad int(+i\_inform - display, -o\_alert - billing\_forwarded - o\_notify))
\end{aligned}
$$

As *int* can return ten different values, we obtain ten different traces t that are overlapping interleavings of the two input traces. All ten traces have the common prefix
$+i\_alert - o\_request + i\_alert - o\_busy + o\_busy$,
their respective endings are:

$+ i\_inform - display - o\_alert - billing\_forwarded - o\_notify$
$+ i\_inform - o\_alert - display - billing\_forwarded - o\_notify$
$+ i\_inform - o\_alert - billing\_forwarded - display - o\_notify$
$+ i\_inform - o\_alert - billing\_forwarded - o\_notify - display$
$- o\_alert + i\_inform - display - billing\_forwarded - o\_notify$
$- o\_alert + i\_inform - billing\_forwarded - display - o\_notify$
$- o\_alert + i\_inform - billing\_forwarded - o\_notify - display$
$- o\_alert - billing\_forwarded + i\_inform - display - o\_notify$
$- o\_alert - billing\_forwarded + i\_inform - o\_notify - display$
$- o\_alert - billing\_forwarded - o\_notify + i\_inform - display$

5.4.2 Overlap

The function $overlap(t_1, ...t_n)$ returns the set of all "overlapping interleaving" sequences that can be generated from the t_i in the input. This includes all those generated from just 2 traces, those from 3 traces, up to those from n traces. We first consider the function, *oi*2, that computes all "overlapping interleavings" of 2 traces and then build up a function for n traces.

Assume that $T(t_1, t_2) = \{t \mid t \text{ is an overlapping interleaving of } t_1 \text{ and } t_2\}$.

*oi*2 returns either an empty set or a set of traces $T(t_1, t_2)$, as follows:

$$
oi2(t_1,t_2) = \begin{cases} \{\} & \text{if } \forall(i,j).(i \neq j) \wedge (\nexists a.(+a \in \alpha_i \vee -a \in \alpha_i)) \rightarrow +a = First(t_j) \\ T(t_1,t_2) & \text{otherwise} \end{cases}
$$

Example 5.4.2 A trivial example overlapping interleaving is:
$oi2(t_{(1)}, t_{(5)}) = \{\}$ Reason: The alphabets are distinct.
However,
$oi2(t_{(6)}, t_{(7)})$ and $oi2(t_{(1)}, t_{(3)})$ are non-empty because the output of $t_{(6)}$ is input to $t_{(7)}$ and $t_{(1)}$ and $t_{(3)}$ have a common input.

Finally we consider the general *overlap* function, operating on n traces ($n \geq 2$). We distinguish two cases:

a) if $n = 2$

$overlap(t_1, ..., t_n) = oi2(t_1, t_n)$

b) otherwise

$$overlap(t_1, ..., t_n) = \left(\bigcup_{\text{for all t}} \left[\bigcup_{i=1}^{n} \{oi2(t_i, t) | t \in overlap(t_1, ..., t_{i-1}, t_{i+1}, ..., t_n)\} \right] \right) \cup \left(\bigcup_{i=1}^{n} overlap(t_1, ..., t_{i-1}, t_{i+1}, ..., t_n) \right)$$

In words, we successively choose all subsets of the given set of traces with exactly one element less. We then compute all overlaps of this subset. In the case of the set containing two elements we have encountered the case where we simply apply $oi2$ to the two traces. If there is more than one trace, we recursively choose all subsets with one element less and compute their overlaps. In the recursive case each trace of the set of computed overlaps is overlapped again using $oi2$ with the trace that was not included in the subset. Thus we obtain the set of all overlaps as required.

5.4.3 Construct

Let t_{ij} be the j-th message of the i-th feature. Recalling that features are sets of traces, $|F_i|$ is the number of traces of feature F_i by use of standard notation.

$$Construct(F_1, ..., F_n) = SingleFeat(F_1, ..., F_n) \cup MultipleFeat(F_1, ..., F_n)$$

$$SingleFeat(F_1, ..., F_n) = \bigcup_{i=1}^{n} F_i$$

$$MultipleFeat(F_1, ..., F_n) = \bigcup_{i=1}^{|F_1|*...*|F_n|*n!} overlap(t_{gj}, ..., t_{hk})$$

We assume the following conditions on the variables: $g \neq h$ and $g, h \in \{1..n\}$ and $j \in \{1..|F_g|\}$ and $k \in \{1..|F_h|\}$.

Example 5.4.3 This example constructs the solution space from the *Calling Number Display* and *Call Forwarding on Busy* features.

$$\begin{aligned} Construct(F_{CND}, F_{CFB}) &= SingleFeat(F_{CND}, F_{CFB}) \cup \\ &\quad MultipleFeat(F_{CND}, F_{CFB}) \\ &= F_{CND} \cup F_{CFB} \cup overlap(t_{(5)}, t_{(7)}) \cup overlap(t_{(6)}, t_{(7)}) \end{aligned}$$

5.5 Application to Running Example

An implementation of this specification has been developed using the Python [Pyt] programming language.

In example 5.5.1 we show some sample output. This example shows the complete solution space for the *Calling Number Display* (CND) and *Call Forwarding on Busy* (CFB) features defined in section 3.2. Note that traces 0 to 9 are the traces belonging to the set $MultipleFeat$ and the remaining are those of the $SingleFeat$. In particular trace 10 is the original trace from the CFB feature, traces 11 and 12 are from the CND feature. Note that neither extraction nor pruning has been applied, hence the complete solution space is returned.

Example 5.5.1

```
    +------------------------ pretty print traces
    |     +------------------- the feature manager
    |     |  +---------------- extraction off
    |     |  |  +------------- pruning off
    |     |  |  |  +---------- trigger event, only relevant if extraction is on
    |     |  |  |  |    + ---- the two features CFB and CND
    |     |  |  |  |    |
>>> print(fm(0, 0, "a", thcfb, thcnd))
These features interact, finding resolution ...

    0 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (+,i_inform,id)(-,o_alert,-)(-,billing_forwarded,D)(-,o_notify,D)
         (-,display,id)
    1 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (+,i_inform,id)(-,o_alert,-)(-,billing_forwarded,D)(-,display,id)
         (-,o_notify,D)
    2 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (+,i_inform,id)(-,o_alert,-)(-,display,id)(-,billing_forwarded,D)
         (-,o_notify,D)
    3 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (+,i_inform,id)(-,display,id)(-,o_alert,-)(-,billing_forwarded,D)
         (-,o_notify,D)
    4 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (-,o_alert,-)(+,i_inform,id)(-,billing_forwarded,D)(-,o_notify,D)
         (-,display,id)
    5 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (-,o_alert,-)(+,i_inform,id)(-,billing_forwarded,D)(-,display,id)
         (-,o_notify,D)
    6 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (-,o_alert,-)(+,i_inform,id)(-,display,id)(-,billing_forwarded,D)
         (-,o_notify,D)
    7 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
         (-,o_alert,-)(-,billing_forwarded,D)(+,i_inform,id)(-,o_notify,D)
```

```
      (-,display,id)
 8 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
      (-,o_alert,-)(-,billing_forwarded,D)(+,i_inform,id)(-,display,id)
      (-,o_notify,D)
 9 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,o_busy,-)
      (-,o_alert,-)(-,billing_forwarded,D)(-,o_notify,D)(+,i_inform,id)
      (-,display,id)
10 :  (+,o_busy,-)(-,o_alert,-)(-,billing_forwarded,D)(-,o_notify,D)
11 :  (+,i_alert,-)(-,o_request,id)(+,i_inform,id)(-,display,id)
12 :  (+,i_alert,-)(-,o_request,id)(+,i_alert,-)(-,o_busy,-)(+,i_inform,id)
      (-,display,id)
```

5.6 Discussion

The specification of the solution space leads us to a clear understanding of the solution space and in particular highlights how traces from individual features can be interleaved. This also provides a picture of the complexity; namely that the solution space increases exponentially in the number of features **and** the length of the interleaved part of the traces. The solution space provides a basis for the development of extraction and pruning methods.

We require two distinct methods to obtain resolutions from the solution space: *pruning* by general rules which removes traces illustrating "bad" behaviour and *extraction* which identifies "good" behaviour. The pruning rules express message patterns, like "two consecutive announcements are not allowed" or "an onhook followed by a number of messages can only be followed by another onhook if the messages contains an offhook". Because these patterns do not relate messages to features, we refer to them as general rules.

We can identify two weaknesses with this specification.

First, the notation is precise but not expressive enough. For example, we require a way to express looping behaviour in some features (as is the case for three way calling when users can toggle several times between partners).

Second, when constructing the overlapping interleaving, we violated our key assumption of not knowing the internal behaviour of features. Thus it is difficult to see how this approach helps to fulfil our aim, namely the development of an online feature manager that can handle black box features.

So, rather than defining pruning methods for this specification, we will reconsider the specification, thus addressing the above weaknesses.

5.7 Operational Specification

When developing the specification of the solution space we have mainly concentrated on *what* the solution space is. However, in the context of the transactional approach it is very important to see *how* we obtain the solution space, that is the operational aspect is very important.

From the conclusions above it is clear that the notation is not suitable. A suitable notation should:

- offer a natural way of expressing events (or messages) as basic concept,
- allow looping behaviour of features to be expressed,
- include (or allow definition of) tree data types (to handle the solution space)

In addition it would be preferable to have some of the following:

- a recognised notation (maybe even standardised)
- tool support for the notation (for simulation and verification)

After considering various formal notations, process algebras seem a natural choice. The basic terms in process algebras are actions which are combined by sequencing and choice operators. Sending or receiving messages can be treated as events. Parallelism operators seem a useful way to describe interleavings. The requirement for complex data types suggests LOTOS [ISO89]. LOTOS combines process algebra with an algebraic data type definition language (Act One). Furthermore, LOTOS is standardised by the International Organization for Standardization (ISO-8807) and well known in the telecommunications area. Extensive tool-sets for verification and simulation are available in the form of CAESAR/ALDEBARAN [CAD] and TOPO/LOLA [LOL].

An extensive case study [Rei01] was performed, in which a LOTOS model of a feature manager and the features from the second contest was developed. The solution space is implemented as a tree datatype and simple extraction rules are defined to reduce the solution space.

LOTOS seems a good choice to perform the task at hand, but several problems were encountered: the model was too large to be handled effectively for verification by the tools. In addition our verification would require reasoning about the tree data type, whereas the available tools are aimed at reasoning about processes and not data types.

Customising the model to handle different numbers of features is not straightforward, as each feature introduces additional events in the feature manager process and several other aspects of the model must be adapted to cope with any given number of features. This was automated by a script, and thus presents only a minor drawback.

Most significantly, reducing the solution space is intended to be implemented exclusively as functions on trees. Pruning adds a significant amount of data type definitions as a complete method for defining regular expressions and finding matches to the same in the solution space is required. Considering that a model containing three features amounted to roughly 30 pages of LOTOS code, half of which was definition of messages and trees not including pruning the additional increase suggests that LOTOS would be unsuitable in this case.

Considering these results, an alternative notation was found in Haskell [Tho99, Has] together with the `Hugs interpreter` [Hug] and the *Glasgow Haskell Compiler* [ghc]. Functional programming fulfils all of our requirements, apart from verification tools. However, functional programming naturally leads to inductive reasoning, so this is not a major drawback. Data type definitions are relatively straight forward in functional programming languages, leading to a clearer model and more efficient prototypes.

We now investigate the improved construction of the solution space using Haskell.

5.8 Haskell Implementation

The construction of the solution space follows Marples approach: The feature manager receives a trigger event which is issued to all features. Responses are collected and fed back to all the features until no further responses are received. During the feedback process the solution space is constructed.

5.8.1 Messages

Messages are defined in the Haskell module `Message.hs`. Three enumeration types provide values for events (type: *Event*), arguments (type *Arg*) and the IO-aspect (type *Io*). Natural equivalence and partial order relations are imposed on the three types (by membership in the respective Haskell classes). Messages consist of a quadruple:

$$type\ Message\ =\ (Io,\ Event,\ Arg,\ Int)$$

where the fourth argument represent the message destination. The destination field is used by the feature manager and the cocoons. A cocoon will ignore all messages where the destination value differs from 0 and its own identity value. This mechanism allows the feature manager to either target particular features ($destination > 0$) or broadcast messages to all features ($destination = 0$).

Example 5.8.1 The following are messages:
(*Rcv*, *Onhook*, *Nil*, 0) and (*Snd*, *Billing_split*, *Splitfactor*, 0).

It is necessary to compare individual parts of a message with fixed values. Projection functions have been defined to extract the appropriate parts of the tuple. Equality tests on values of type *Message* are inherent.

Messages can be partitioned into feature messages and transaction control messages. The former are messages describing the intended behaviour of features and have been described in detail in Chapter 3. The latter, of which there are 4, are used by the feature manager to control the transactional cocoons. The functions *is_tcmsg* and *is_fmsg* tell us which class of messages a message belongs to.

5.8.2 Features and Cocoons

Features and cocoons are defined in the module `Features.hs`. Recall that features are extensions to the basic behaviour of a system. We treat the basic system in the same manner as other features. Hence we essentially see features as a black boxes that react to a trigger event by providing a (possibly empty) response. Features are modelled as functions, and initial and final state of a transaction are passed as values. However, as this information will not be available in the operational system, the feature manager does not make use of it. Thus, the requirement of features being treated as black boxes is satisfied.

Definition 5.8.1 (Feature) A feature is a function:
featbehave :: *Int* → *Message* → *Int* → (*Int*, *Queue Message*)
such that *featbehave s m f* is the new state and the response of feature *f* in state *s* to message *m*.

Example 5.8.2 *featbehave* 1 (*Rcv*, *Dial*, *Nil*, 0) 1 leads to a call of *tl* 1 (*Rcv*, *Dial*, *Nil*, 0), thus evaluating the behaviour from a teenline feature (which in the implementation is feature 1).

featbehave returns pairs of the new state and the response of a feature. Note that features can be seen to have moved into a new state in their automata representation (or not), as well as produce an empty response or not. All combinations are possible:

Example 5.8.3
tl 1 (*Rcv*, *Dial*, *Nil*, 0) ⇝ (2, *enqueue emptyQueue* (*Snd*, *Announce*, *Wrongpin*, 0))
bcs 1 (*Rcv*, *I_alert*, *Nil*, 0) ⇝ (1, *enqueue emptyQueue* (*Snd*, *O_busy*, *Nil*, 0))
ct 7 (*Rcv*, *I_disconnect*, *Nil*, 0) ⇝ (0, *emptyQueue*)
tl 2 (*Rcv*, *Connect*, *Nil*, 0) ⇝ (2, *emptyQueue*)

Features are encapsulated in a transactional cocoon. The cocoon is the point of interaction of the feature manager with the feature. More importantly the cocoon controls the transactional mechanism that is used to explore possible behaviour by polling features reactions in order to construct the solution space. The control of the

construction process is part of the feature manager process. Each cocoon is associated with a given feature throughout the runtime of the system.

Definition 5.8.2 (Cocoon) A cocoon is a function:
$cocoon :: Int \rightarrow Stack\ Int \rightarrow Int \rightarrow Int \rightarrow Message \rightarrow Bool \rightarrow (Queue\ Message, CState)$
such that *cocoon cs rb fid co m f* is the new state of a cocoon and the response of the associated feature obtained by *featbehave cs m fid*.

Definition 5.8.3 (State of a Cocoon) The state of a cocoon is a 4-tuple
$CState = (Int, Stack\ Int, Bool, Int)$
containing the features current state, a rollback stack, a flag identifying whether the cocoon is in playback mode and the identity of the associated feature.

Rather than defining each of these elements more precisely we describe the behaviour of the cocoon. Upon receipt of a message the cocoon determines whether a transaction control message or a feature message has been received and whether the received message has indeed been targeted at this cocoon. If the message is not targeted at the cocoon it will simply be ignored, resulting in an empty queue and the current state of the cocoon being returned.

When a feature message is received, the cocoon may or may not be in playback mode. A cocoon in playback mode has the role of re-instantiating a certain feature state (usually after the feature manager has made a decision as to which features shall be executed and which blocked). In this mode no responses from the feature are collected (they will have been obtained earlier and are part of the stored resolution). Thus an empty queue and a new state of the cocoon are returned. If, more interestingly, the cocoon is not in playback mode, the return value is the response of the feature and the new cocoon state.

Upon receipt of a transaction control message if the cocoon is in playback mode and the control message is a *Commit_transaction* message, then playback will have finished and an empty queue and a new state are returned. If we are not in playback there are four distinguished cases, dependent on the control message (we use the same messages as were advocated by Marples [Mar00]). All four cases return an empty feature response as the features play no role in this process. The state of the cocoons changes and the new state is returned.

- *Start_transaction*: A new transaction is started, and the current state of the cocoon is saved on the rollback stack.
- *Abort_transaction*: We are no longer interested in the whole series of transactions, hence the new state becomes the state of the cocoon before the first transaction: the bottom element of the rollback stack.

- *Rollback_transaction*: We want to go back one level, i.e. undo the last transaction. The new state is obtained by popping the last element from the stack and making this the new state.
- *Commit_transaction*: Committing a transaction requires the cocoon to be returned to its initial state. This can be achieved performing the same action as for *Abort_transaction*. Because the new state is not returned from *construct* the abort message is not actually sent. The cocoon enters playback mode, and the feature manager will now play back the messages so that features can be advanced to the correct states.

5.8.3 Feature Interaction

The earlier definition of feature interaction (Definition 5.3.4) must be slightly adapted to be useful within a Haskell setting. We do not have the same knowledge about features as was assumed earlier, namely we do not know the complete set of traces of a feature. The only information available is whether a feature reacts to a trigger event or not. The definition of feature interaction is equivalent to the intuitive understanding of Definition 5.3.4:

Definition 5.8.4 (Feature Interaction) Features interact iff

- more than one feature has responds to a trigger message, or
- a response is received upon feeding a response back to the features.

The first case simply means that more than one cocoon has returned a non-empty queue as response to a feature message. The second case means that a feature has responded with a message that acts as trigger to another feature. Note that these two cases can occur in any order, between any number of features and in any quantity. For example the original trigger might just produce one response, but the feedback thereof leads to several features responding. The new feedback can again produce more responses. This process could be potentially infinite (call forwarding loops in systems with single point of call control for example) so an operational system would require the amount of feedback allowed to be restricted. We return to this issue in Chapter 7.

Note that no judgement about the desirability of the interaction is made at this point: the responses can clash thus leading to inconsistent behaviour or they can be complimentary and co-exist without any problems. Pruning will handle inconsistent behaviour and we investigate this later.

5.8.4 The Feature Manager

The role of the feature manager is to detect and resolve interactions, as specified earlier. In addition, the feature manager has to control the transactional cocoons as part of the feedback process and the rollback and commit mechanism.

Definition 5.8.5 (Feature Manager) The feature manager is a function
fm :: [*CState*] → *Message* → *RegExpr Message* → *Augtree Message*
such that *fm fs m rules* is a resolution to any potentially detected interactions observing the rules *rules* in a system with features fs.

The feature manager is implemented as :

$$fm\ fs\ m\ rules\ =\ extract\ (prune\ (extractdup\ (construct\ fs\ m))\ rules)$$

The functions *extract*, *extractdup* and *prune* which identify a resolution in the solution space are to be refined in Chapter 6. The next two sections will address the problem of committing a resolution and the *construct* function. The feature manager and all listed subfunctions are defined in the Haskell module `Main.hs`.

5.8.5 Committing a Resolution

Assuming that we have identified a single resolution (the solution space consists of a tree with a single leaf node), we need to consider how this resolution is committed to the system.

In the process of constructing the solution space, features are queried repeatedly about responses to messages and we don't know which state they are currently in. What we do know however, is that we can return them to their former state (i.e. the state they had before the trigger event was received). Committing the chosen resolution involves sending the respective responses to the user and/or remote switch and updating some features to a new state.

The function *commit* has the purpose of doing this:

$$commit\ ::\ Augtree\ Message\ \rightarrow\ [CState]\ \rightarrow\ ([Message], [CState])$$

The commit process is initiated by the feature manager issuing a *Commit_transaction* message to all features and is terminated by a second such message (we could have used a different message, but decided to not introduce any further messages).

5.9 Construction of the Solution Space

The complex construction process has been split into several functions.

$$construct\ ::\ [CState]\ \rightarrow\ Message\ \rightarrow\ Augtree\ Message$$

is the main function. The solution space is constructed from a list of current cocoon states and a message. Figure 5.3 provides an overview of sub-functions. We briefly describe the purpose of the individual functions before describing the most interesting one, *feedback* in more detail.

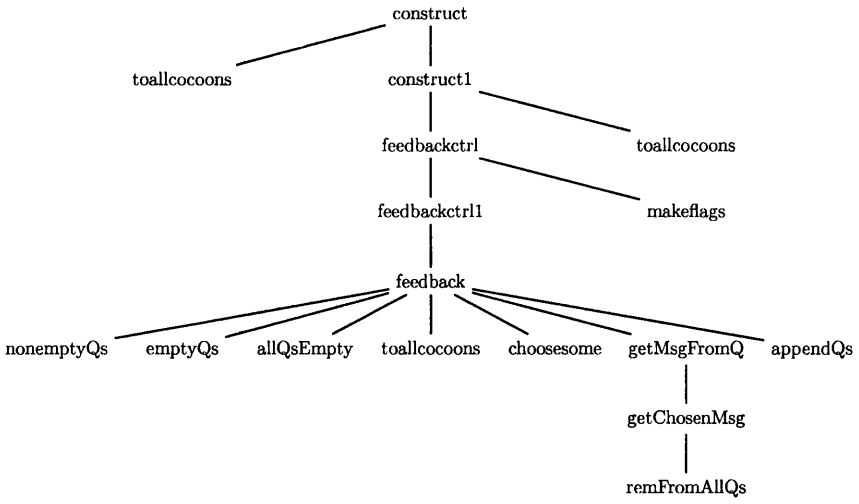

Fig. 5.3: Hierarchy of Functions in the Construction Process

construct returns the solution space obtained by *construct*1 and initiates the transaction management.

*construct*1 gathers responses to the trigger event and then initialises the feedback process. It also inserts the trigger message as root in a new tree.

toallcocoons enables a message to be passed to all cocoons and then for the responses to be returned.

feedbackctrl assures that feedback is run the required number of times with the correct arguments. Depending on the number of features different cases have to be explored. In particular we must explore all branches involving 1 feature, then all those involving any combination of 2 features, and then all those with 3 and so on. This is achieved by passing the list of flags, generated by *makeflags* to *feedbackctrl*1 which controls the actual feedback process.

makeflags guarantees that all permutations of features are explored. A list of multibit-flags is generated. Each flag represents a feature being enabled or disabled. The case of all features being disabled is obviously not interesting, and so is not considered. For example, the set of flags for 2 features is $[[True, False], [False, True], [True, True]]$.

feedback controls the rollback process, initiates the feeding back of messages to features and inserts responses into the tree. We will consider this process in more detail in Section 5.9.1.

nonemptyQs counts the number of non-empty queues in a list of queues.

emptyQs returns a list of empty queues, it basically empties all queues in the input list.

allQsEmpty reports whether all queues in a list are empty.

choosesome allows one to disregard replies from some features. As we cannot physically disable the features, it seems easiest to simply ignore their responses.

getMsgFromQ controls the extraction of a message from a particular queue.

getChosenMsg extracts a message from the head of a particular queue.

remFromAllQs removes a particular message from all queues in which it occurs as front element.

appendQs appends the content of queues in one list to the queues at the same position in a second list to obtain a list of merged queues.

To summarise, *construct* initiates the whole construction process, a new tree is constructed with the trigger message as root node. Then all settings of different features are explored by *feedback* whereby *feedbackctrl* ensures that all possible settings have been considered. The resulting solution space contains the branches that have been added by the individual feedback processes. Note that this can lead to duplicate subtrees. For example, consider a system containing 3 features f_1, f_2 and f_3, in which only f_1 and f_2 react to the trigger event. In this setting we would expect that the traces generated with all three features enabled are the same as those with only f_1 and f_2 present. Traces with 2 features present one of which is f_3 will be equal to those with just the other feature present.

5.9.1 Feedback

Feedback has to ensure that all possible interleavings of the responses are explored. We have decided on a fine grained interleaving, that is any order of the responses must be explored, independent of the feature which sent them. Responses will be added to the message queues and it is irrelevant whether responses in the queues occurred as result of the first trigger or of any subsequent feedback.

Recall that all combinations of any number of features have to be explored, including those involving only one feature.

Feedback has been split into two parts: *feedbackcontrol* which controls the exploration of the different combinations of activated features, and *feedback* which ensures the correct construction of a tree with a certain number and combination of enabled features .

feedback is a function that takes a multibit flag representing the enabled status of the features, a list of message queues (one queue per feature), the current state of the features, the solution space constructed so far, an integer representing how many branches have been explored so far and a state stack. The *feedback* function returns the solution space with any new solutions inserted together with the new state of

the features. The reason for the inclusion of the state stack as input may not be immediately obvious. It is used to store state information and contains the old message queues as well as how many branches have been explored starting from these queues. Note that the stack does not store the state of the features (this information is handled by the cocoons).

$$\begin{aligned} \mathit{feedback} :: [\mathit{Bool}] \rightarrow [\mathit{Queue\ Message}] \rightarrow [\mathit{CState}] \rightarrow (\mathit{Augtree\ Message}) \rightarrow \mathit{Int} \\ \rightarrow \mathit{StateStack} \rightarrow (\mathit{Augtree\ Message}, [\mathit{CState}]) \end{aligned}$$

such that *feedback bs qs fs t c* is the solution space with all traces obtainable from *qs* and the enabled *fs* inserted.

feedbackctrl calls *feedback* several times, once for each combination of enabled features (as represented by the list of multibit flags). The variable c denotes the number of currently explored branches, and is initially set to 0.

Let us now consider the details of *feedback*. We distinguish 4 cases:

1. All message queues are empty and the state stack is empty,
2. All message queues are empty and the state stack is non-empty,
3. At least one message queue is non-empty and c is less than the number of non-empty message queues,
4. At least one message queue is non-empty and c is equal to the number of non-empty message queues.

An example will illustrate these cases. Assume a system with three features, starting from a point in the system where the features have received a trigger and the message queues containing the responses are: $Q1 = [a, b, c]$, $Q2 = [b, d]$ and $Q3 = [e]$. The value of c is 0, i.e. no branches have been explored yet. Further assume that we are currently interested in the case were all three features are active. Let $C1$, $C2$ and $C3$ represent states of the three cocoons, s_0 the current state stack and t_0 the solution space constructed so far. Feature 3 reacts to receiving a message a with the response f and no other features respond to any message. For the changes to the stack and tree we refer to Fig 5.4 and its continuation in Fig. 5.5. The bottom half of the figures represents the stack, the top half the tree. The node labelled `M` represents the insertion marker.

Example 5.9.1
feedback $[T, T, T]$ $[[a, b, c], [b, d], [e]]$ $[C1, C2, C3]$ t_0 0 s_0

This is an instance of the third case: there is **at least one non-empty message queue and c is less than the number of non-empty message queues.**

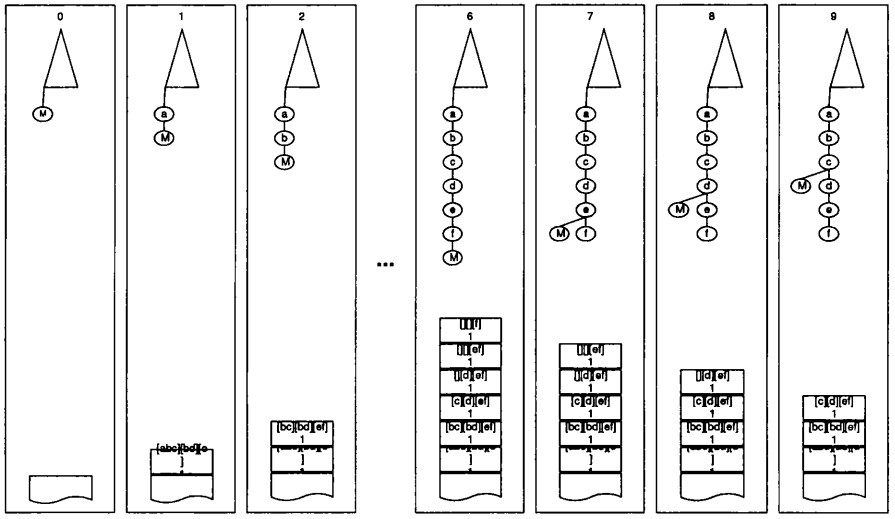

Fig. 5.4: Feedback: Solution Space and Statestack 1

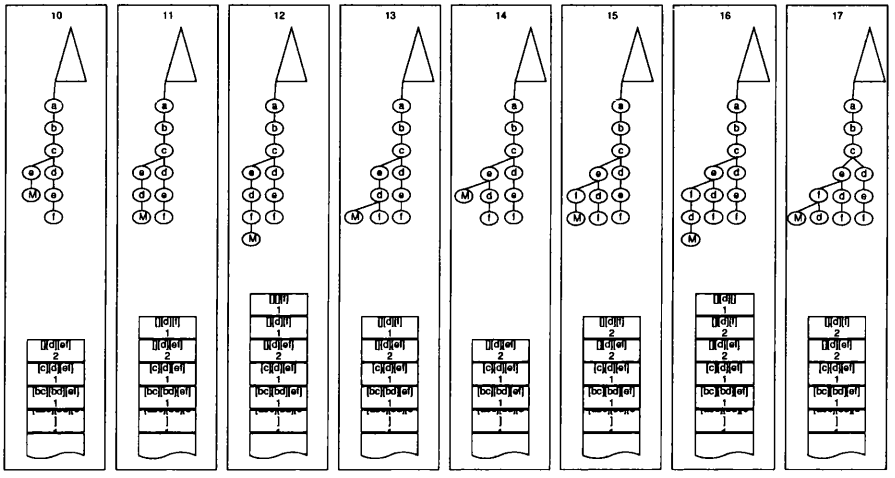

Fig. 5.5: Feedback: Solution Space and Statestack 2

We are required to pick one message from a non-empty queue, insert this message into the tree and feed it back to the features. Before the message can be fed back, a new *start_transaction* message needs to be sent to instruct the cocoons to save all respective information for a potential rollback. The new responses must be collected and added to the message queues. Feedback will be recursively performed after the current state has been stored on the stack.

Because $c = 0$, the message must be chosen from the first non-empty queue, giving us the following:

Example 5.9.2
feedback $[T, T, T]$ $[[b, c], [b, d], [e, f]]$ $[C1, C2, C3]$ $t1\ 0\ s1$
Note that Feature 3's response f has been added to the third queue.

Clearly we can continue in the same way for some more steps:

Example 5.9.3
feedback $[T, T, T]$ $[[c], [d], [e, f]]$ $[C1, C2, C3]$ $t2$ 0 $s2$
Note that b has been removed from all queues where it occurred as head, thus implementing the overlap as discussed in section 5.4. The feedback process continues as follows:

feedback $[T, T, T]$ $[[], [d], [e, f]]$ $[C1, C2, C3]$ $t3$ 0 $s3$
feedback $[T, T, T]$ $[[], [], [e, f]]$ $[C1, C2, C3]$ $t4$ 0 $s4$
feedback $[T, T, T]$ $[[], [], [f]]$ $[C1, C2, C3]$ $t5$ 0 $s5$
feedback $[T, T, T]$ $[[], [], []]$ $[C1, C2, C3]$ $t6$ 0 $s6$

We are now in a position where **all message queues are empty and the state stack is non-empty** (case 2). No feature has responded to the most recent event (either the trigger or a fed back response) and all previous responses have already been fed back. However we can rollback to a previously stored state by popping the topmost state from the stack. We then reinstantiate feedback.

To rollback, the state of the message queues and number of explored branches are as popped from the state stack. The cocoons are sent a rollback message so that they can restore previous states of the features. It only remains to shift the insertion point in the tree: *movemarker* moves the `InsMarker M` leaf up one level in the tree so that it becomes a child of its current grandparent.

The `InsMarker M` denotes the point at which the next node should be inserted. This insertion point is always the leftmost leaf of the leftmost branch (a decision made for efficiency reasons; accessing elements at the start of a list is fastest in Haskell). During rollback, if the insertion point is the root of the tree, we cannot move any further up, so we are done. If the insertion point is a leaf of the root node again we cannot move up. If the insertion point is at least two levels down in a tree we have to distinguish whether it is exactly two levels down or whether it is even further away. In the latter case we recursively call *movemarker* to have the marker moved. If it is exactly two levels down from node n (this can also be the root node), we insert the marker as a new child of n and remove the marker from its previous location.

Now, we call feedback again:

Example 5.9.4
feedback $[T, T, T]$ $[[], [], [f]]$ $[C1, C2, C3]$ $t7$ 1 $s7$

Now **at least one message queue is non-empty and c is equal to the number of non-empty message queues** (case 4). We have explored all possible behaviours from the current position and simply need to rollback further. This is achieved by recursively calling *feedback* with the current arguments ensuring that the queues are emptied first. This results in the following sequence of calls:

Example 5.9.5
feedback $[T, T, T]$ $[[], [], []]$ $[C1, C2, C3]$ $t7$ 1 $s7$
feedback $[T, T, T]$ $[[], [], [e, f]]$ $[C1, C2, C3]$ $t8$ 1 $s8$
feedback $[T, T, T]$ $[[], [], []]$ $[C1, C2, C3]$ $t8$ 1 $s8$
feedback $[T, T, T]$ $[[], [d], [e, f]]$ $[C1, C2, C3]$ $t9$ 1 $s9$

At this point c is again less then the number of non-empty queues, which means that there is a possible interleaving that has not been explored. Since $c = 1$ this interleaving is obtained by picking the message from the second non-empty queue. This results in the following sequence of calls:

Example 5.9.6
feedback $[T, T, T]$ $[[], [d], [f]]$ $[C1, C2, C3]$ $t10$ 0 $s10$
Note that we entered a new level with no possibilities explored, hence c is again 0.
feedback $[T, T, T]$ $[[], [], [f]]$ $[C1, C2, C3]$ $t11$ 0 $s11$
feedback $[T, T, T]$ $[[], [], []]$ $[C1, C2, C3]$ $t12$ 1 $s12$
feedback $[T, T, T]$ $[[], [], [f]]$ $[C1, C2, C3]$ $t13$ 1 $s13$
feedback $[T, T, T]$ $[[], [], []]$ $[C1, C2, C3]$ $t13$ 1 $s13$
feedback $[T, T, T]$ $[[], [d], [f]]$ $[C1, C2, C3]$ $t14$ 1 $s14$
feedback $[T, T, T]$ $[[], [d], []]$ $[C1, C2, C3]$ $t15$ 0 $s15$
feedback $[T, T, T]$ $[[], [], []]$ $[C1, C2, C3]$ $t16$ 0 $s16$
feedback $[T, T, T]$ $[[], [d], []]$ $[C1, C2, C3]$ $t17$ 1 $s17$

This process continues until **all message queues are empty and the state stack is empty**. This is the case 1 and arises when no feature has responded to the most recent event and all previous responses have already been fed back. The empty stack indicates that we cannot rollback any further. Consequently, we know that we have finished and hence all that remains is to return the tree and the current state of the features.

5.10 Application to Running Example

So far we have described the construction of the solution space in the Haskell implementation by showing isolated parts and explaining some of those with the help of small examples. In this section we will show a more complex construction to provide the complete picture.

Recall the implementation of the feature manager:

$$fm\ fs\ m\ rules\ =\ extract(prune(extractdup(construct\ fs\ m))\ rules).$$

For the purpose of this example we assume that *extract*, *prune* and *extractdup* are identity functions (i.e. we only want the feature manager to construct the solution space).

```
main = fm [(0, emptyStack, False, 2),        — Call Forwarding Busy in State 0
           (1, emptyStack, False, 7)]        — Calling Number Display in State 1
           (Rcv, I_alert, Nil, 0)            — I_alert as trigger message
           (MkRE (Snd, Onhook, Nil, 0))      — dummy, pruning is disabled
```

This example call constructs the solution space for the *Call Forwarding Busy* and the *Calling Number Display* feature. CND is in a state where it responds to *i_alert* by producing a trigger for CFB. The expected output is a tree containing two solutions: one solution representing the interleaved behaviour and one for the behaviour of CND (i.e. when CFB is disabled). The case when CND is disabled does not lead to the construction of a solution, as CFB does not react to the provided trigger.

```
hugs > main

((Rcv,I_alert,Nil,0),{0})[((Snd,O_busy,Nil,0),{1})
                          [((Snd,O_alert,Nil,0),{2})
                             [((Snd,Billing_forwarded,Nil,0),{2})
                                [((Snd,O_notify,Nil,0),{2})[]]]]
                          ((Snd,O_busy,Nil,0),{1})[]
                         ]
```

The output is illustrated in Fig. 5.6.

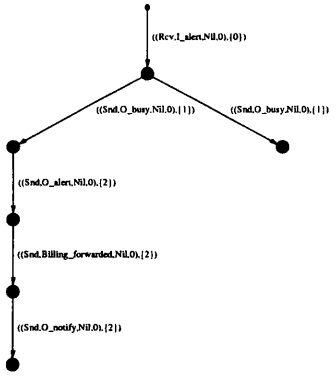

Fig. 5.6: Solution space constructed for CND and CFB using Feedback

Note that the output differs marginally from the result produced in section 5.5. These differences occur because we only have a very restricted view of the feature behaviour in the implementation whereas in the specification a complete knowledge was assumed.

5.11 Summary

We introduced the notions of solution, solution space and resolution. A specification and implementation for the construction of the solution space was completed with applications to the running example.

Chapter 6

Resolution

6.1 Introduction

In the previous chapter we constructed all possible **solutions**. However, recall that solutions are not necessarily **resolutions** for a feature interaction. Recall that a resolution is a solution that does not violate any of the specified properties. Hence it remains to remove all those that don't form resolutions from the solution space giving us a **resolution space**. We can then extract the best resolution.

In this chapter we discuss how we can distinguish bad and good solutions, and also how we find the best resolution. The term *pruning* is used to describe operations that remove bad solutions, operations identifying the best resolution are referred to as *extraction*. Each operation is based on a rule which has been derived empirically by analysis of examples. Application order is significant.

We consider how resolution fits into the overall runtime approach. A simple composition of construction and pruning is briefly considered, followed by a more efficient on-the-fly approach.

6.2 Identifying Resolutions

We base the operations of pruning and extraction on rules. The purpose of the rules is to discriminate between bad and good solutions and also to describe the quality of resolutions. Recall that the rules are identified by an offline analysis of the behaviour of the model, as described in sections 4.2 and 4.3.

Rules can take different forms and this section provides only an overview, details are explored in the following sections. Two classes of rule can be distinguished: message dependent rules and message independent rules. Examples describing concrete good or bad behaviour constitute a further class; however, they are of little interest, and hence we will not consider them further.

Message independent rules can include concepts such as priorities of features and the satisfaction of the maximal number of features. A feature is considered to be satisfied when its intended behaviour can be exhibited, and clearly it is best to choose a resolution that satisfies the largest number of features. Message independent rules are used for *extraction* of the best resolution.

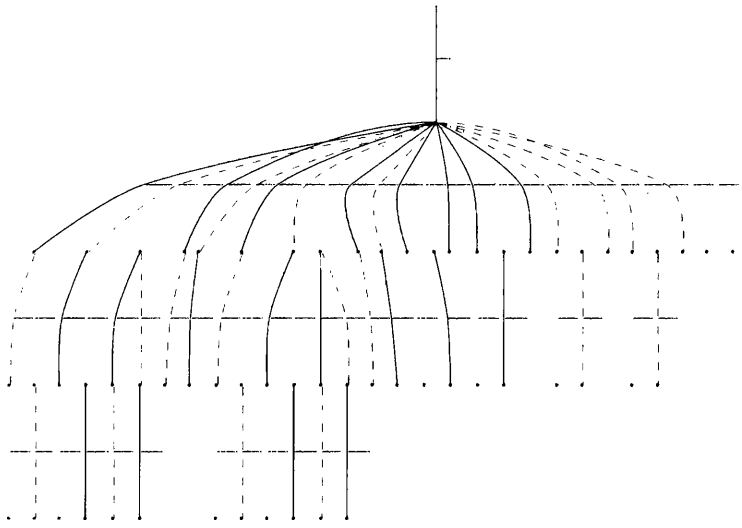

Fig. 6.1: The Constructed Solution Space for Example 6.3.1

Message dependent rules can be seen to be more powerful, but they also require more information. In order for message dependent rules to be useful, a semantics on the messages is required. This allows one to develop relations on messages, such as a class of treatments, or a class of billing messages. This understanding enables us to build grammars describing good or bad behaviour, for example "a treatment following an onhook message is not useful unless there was an offhook in between". Message dependent rules are used for *pruning* the solution space.

In section 6.4 we will consider extraction rules and in section 6.5 we will discuss pruning and its implementation.

6.3 An Example

Throughout this section we consider the following example to illustrate the application of the different rules.

Example 6.3.1 Assume the presence of four features in the system: Teenline, Split Billing, Reverse Charging and Terminating Call Screening. Teenline is initially in state 1, the others are in state 0. The trigger event is *dial*. From the feature definitions (Appendix A) we know that the first three features react to a *dial* trigger in the given states, TCS does not react to this trigger. The respective responses are: *announce(wrongpin)*, *billing_split(splitfactor)* and *billing_reverse*. Feedback does not lead to any further responses.

Figure 6.1 gives an indication of the complete solution space as constructed by the method described in Sections 5.8 and 5.9. Note that throughout this chapter we have labelled the branches rather than the nodes of the trees, thus the figures deviate slightly from the actual solution space data structure – we assume the relation to be

obvious. As the labels are barely visible, the tree has been colour and pattern coded: branches labelled by the same messages are assigned the same colour and pattern. We will use this colour/pattern encoding throughout the remainder of this chapter. The assignment of colours and patterns to messages is as follows:

announce(wrongpin)	blue	dotted
billing_split(splitfactor)	green	solid
billing_reverse	red	dashed

The implementations of *fm* and *extractTree* have been adapted to the particular rule that we wish to apply. We will demonstrate these implementations with respect to four different rules in the subsequent sections.The function *fm fs m rules* is initialised with the following parameters to encode the example: $fs = [(1, emptyStack, False, 1), (0, emptyStack, False, 3), (0, emptyStack, False, 4), (0, emptyStack, False, 5)]$ and $m = (Rcv, Dial, Nil, 0)$. The *rules* parameter is only relevant when pruning is applied.

6.4 Message Independent Rules – Extraction

Message independent rules are characterised by not requiring any information about the semantics of the messages. Thus, they can be applied to a solution space without any knowledge of the messages occurring therein. However, it is assumed that a comparison between messages is possible and that we are able to differentiate between messages from different features (i.e. we know which messages originated from the same feature).

6.4.1 Duplicates

In Section 5.9 we have mentioned that the solution space might contain several duplicate branches. Consider the example again: assume 3 features (f_1, f_2, f_3) two of which (f_1 and f_2) respond to the same trigger. It is to be expected that a branch corresponding to this trigger with all three features active is identical to one with feature f_3 disabled. Furthermore any branch with two features active, one of which is f_3, is identical to the branch with just the other feature active.

Rule 1 *Duplicate subtrees sharing the same parent can be removed.*

Removal of duplicates as been implemented as a function on the tree datatype:

$$extractdupTree :: Ord\ \alpha \Rightarrow Augtree\ \alpha \rightarrow Augtree\ \alpha.$$

The recursive implementation is rather obvious. Note, that if a tree is an `InsMarker` – denoting the point for the next insertion – it can be replaced with an `EmptyTree`

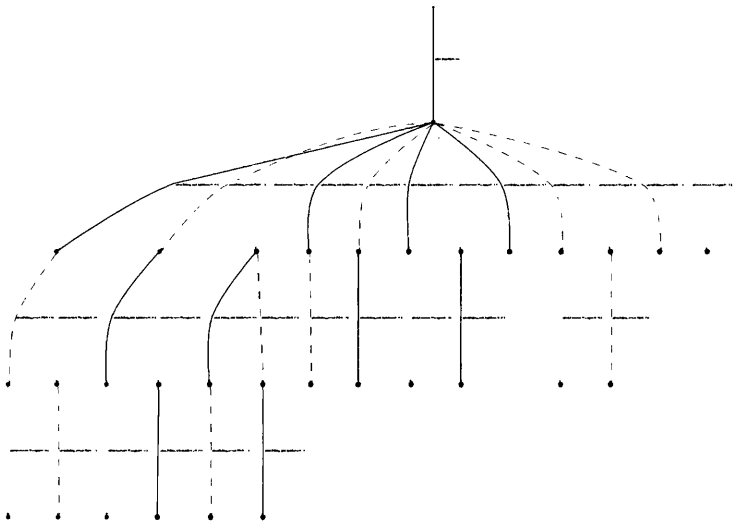

Fig. 6.2: The Constructed Solution Space for Example 6.3.1 with Duplicates Removed

as we do not wish to perform further insertions (recall that the solution space has been completely constructed[1]). Applying *extractdupTree* operation to the example tree (Fig. 6.1), we can reduce the tree to Fig. 6.2. The respective equation of *fm* is: *fm fs m rules* = *extractdup*(*construct fs m*).

Obviously a duplicate branch will not provide a new solution, so all duplicates can safely be removed. Hence, Rule 1 is considered safe. It is advisable to apply this rule straight after the construction is completed in order to reduce the complexity of the tree and make the other, subsequently applied, operations more efficient.

6.4.2 Satisfying Features

A feature is satisfied by a trace when its intended behaviour is exhibited. So, if a feature f_1 responds with messages a_1, a_2 and a_3 in that order, every trace in the tree containing these messages in *that order* satisfies Feature f_1.

The construction of the solution space maintains the relative order of messages as intended by a feature. However, the messages can be arbitrarily interleaved with responses from other features. If feature f_2 responds with b_1 and b_2 then some possible traces satisfying both f_1 and f_2 are $a_1.a_2.a_3.b_1.b_2$ and $a_1.b_1.b_2.a_2.a_3$, but $a_1.b_2.b_1.a_3.a_2$ is not acceptable.

A user subscribing to a number of features would expect all to work. We have seen that this is not always possible (by definition of the feature interaction problem), but we want to satisfy as many features simultaneously as possible.

[1] The `InsMarker` could also be replaced at the end of the construction phase

Rule 2 *Traces containing messages from the largest possible number of features are better.*

Again, this rule has been implemented as an operation of the tree data type. Note that the tree datatype carries information about the origin of each message, as we have indicated earlier, i.e. every node contains the message and a number identifying the originating feature.

The implementation requires two passes through the tree. In the first pass a list containing the number of different features in each trace (starting at the root and ending in a leaf) is constructed. The second pass removes all branches that lead to a leaf for which the number of features along the trace is less than the maximum value in the list.

Let us consider the implementation in slightly more detail. The multiple passes required are represented by the application of several functions. In particular the function $exmostsat1 :: Ord\ \alpha \Rightarrow Set\ Int \rightarrow AugTree\ \alpha \rightarrow [Int] \rightarrow [Int]$ produces the list of features satisfied along all traces. $exmostsat2 :: Ord\ \alpha \Rightarrow Augtree\ \alpha \rightarrow [Int] \rightarrow Augtree\ \alpha$ removes the unwanted branches. The main function uses the two subfunctions to perform the extraction of the branches with the most satisfied features:

$$exmostsat :: Ord\ a \Rightarrow Augtree\ \alpha \rightarrow Augtree\ \alpha$$
$$exmostsat\ t = exmostsat2\ t\ (exmostsat1\ emptySet\ t\ [])$$

Applying this operation to the solution space of example 6.3.1 (Fig. 6.1), we can reduce the tree to that shown in Fig. 6.3. The respective definition of *fm* is: $fm\ fs\ m\ rules = extract(construct\ fs\ m)$ where $extractTree\ t = exmostsat\ t$.

In contrast to removing duplicate branches, this extraction method is unsafe, as it removes resolutions.

Due to the unsafe nature of this operation an application to the unpruned tree is not advisable. Consider for example a tree which contains a trace involving messages from 4 different features, whereas all other traces involve less features. Clearly the trace satisfying 4 features will be the only one retained after applying Rule 2. Now, consider that this branch may contain conflicting messages, i.e. it represents unwanted behaviour, which means we have not been able to find a resolution. From this example we deduce that Rule 2 must be applied after the pruning process (see section 6.5), i.e. to the resolution space rather than the solution space.

6.4.3 Priorities

A simple, but nevertheless effective method of extraction is prioritising features. We do not possess information about features' identities per se, but we do know their

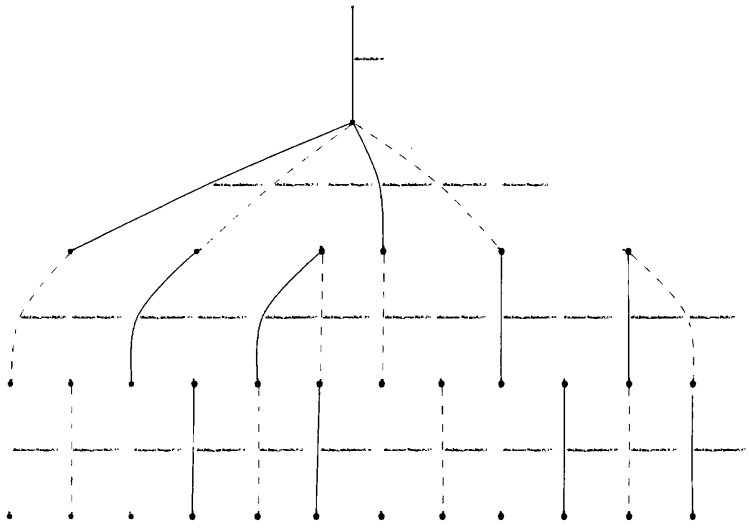

Fig. 6.3: The Constructed Solution Space for Example 6.3.1 after Extracting Traces Satisfying most Features

relative position in the network. For example, we know that a message has been received from feature f_i, but we do not know the identity of feature f_i (i.e. whether f_i is *call waiting*, *three way calling* or any other feature). We refer to i as the features *connection number*.

A simple precedence scheme is as follows: Let features with a low connection number have higher priority. Assume that the lowest connection number is 1. This definition implies that trace i has precedence over trace j if trace i satisfies feature 1. We can continue this by saying that from the remaining traces those satisfying feature 2 are preferable to all remaining ones and so on.

Rule 3 *Traces satisfying features with the highest priority are preferable.*

Clearly this scheme could be extended to a system of weighted priorities in which each feature has an associated weight (where the features with the highest weight are preferable). Each trace would then have a weight equal to the sum of the weights of the features satisfied by that trace. The trace with the highest weight would be preferred.

We have implemented these two ways of prioritising features, extraction by priority by connection number and extraction by priority by weight. The respective function declarations are given below and the results of applying these versions of the rule to the example can be seen in Figs. 6.4 and 6.5:

$$expriobynumber :: Ord\ \alpha \Rightarrow Augtree\ \alpha \rightarrow Augtree\ \alpha$$

$$expriobyweight :: Ord\ \alpha \Rightarrow Augtree\ \alpha \rightarrow [Int] \rightarrow Augtree\ \alpha$$

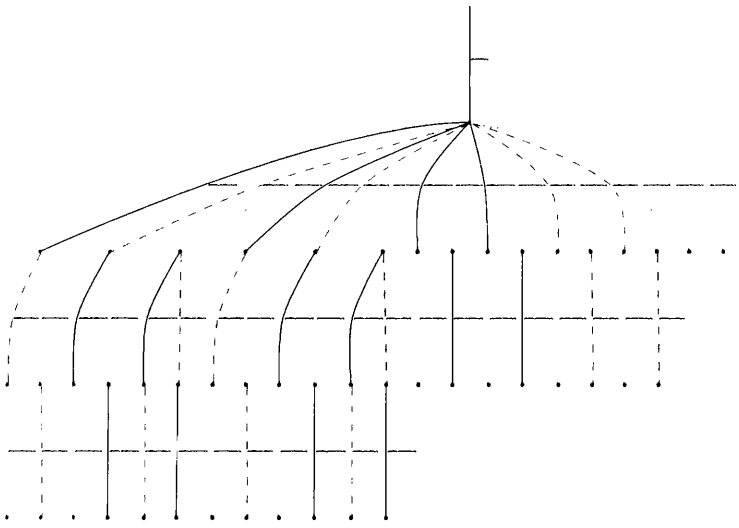

Fig. 6.4: The Constructed Solution Space for Example 6.3.1 after Extraction by Priorities by Connection Number

fm is implemented as *fm fs m rules* = *extract* (*construct fs m*) and *extractTree* is implemented as

extractTree t = *expriobynumber t* and
extractTree t = *expriobyweight t* $[0, -1, 5, 2, 2]$
respectively. The list specified in the *extractTree t* = *expriobyweight* contains the weight for the input trigger event (a 0 in our example, i.e. the case where the destination field in messages is 0) followed by the weights of the individual features.

This extraction method is unsafe – once traces are removed all remaining traces may in fact not be resolutions. Indeed this situation arises when the weighted extraction method is applied to example 6.3.1. All retained traces contained two contradictory billing messages and hence clearly are not resolutions.

6.4.4 Choosing One Resolution

A "best" resolution is not necessarily unique. Suppose that after removing duplicates and pruning we have already extracted resolutions which satisfy most features and applied a priority scheme, but a tree still remains with more than one trace. At this point we have found more than one resolution that we would classify as the best, but obviously the system can only commit to one trace. However, if both traces represent behaviour that from a qualitative point of view is indistinguishable, we can simply choose one.

Rule 4 *If there are a number of "best" resolutions, choose one.*

Note that it would be preferable from the user's point of view if this is a deterministic choice. The user is not (and shall not be required to be) aware

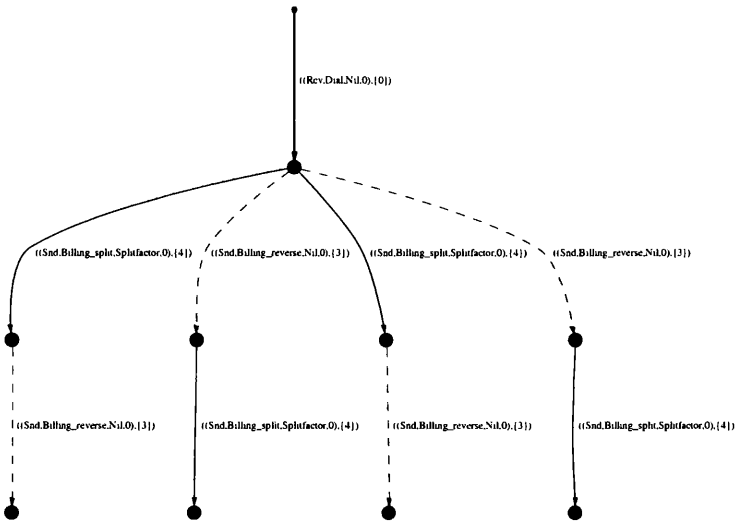

Fig. 6.5: The Constructed Solution Space for Example 6.3.1 after Extraction by Weighted Priorities

of the resolution process and the presented behaviour must be consistent across a number of separate calls. We have implemented this rule such that it simply chooses the leftmost branch of the tree, mainly because this is most efficient with respect to our implementation. The definition of the corresponding function is given by: $expickone :: Ord\ \alpha \Rightarrow Augtree\ \alpha \rightarrow Augree\ \alpha$.

Applying this rule to our example solution space, we obtain the solution space shown in Fig. 6.6. The respective definition of *fm* is: $fm\ fs\ m\ rules = extract(construct\ fs\ m)$, where $extractTree\ t = expickone\ t$.

6.5 Message Dependent Rules

A semantics of messages allows us to define more sophisticated rules, we will consider these now. While it might seem unreasonable to have a semantics of messages, especially in a setting where the internal behaviour of the features is unknown, some knowledge about the semantics of messages is practical due to the message set in telephone switching systems being very restricted. New features cannot simply introduce new messages, they rather need to rely on the existing ones. Clearly, the user requires consistent meaning of the signals received.

Messages can be grouped in classes. These include the rather obvious classes "billing messages", "user messages" and "system messages" – similar to Chapter 3. In addition, and more interesting, we can have classes like "announcements", "treatments", "tones", "hookevents".

Classes of messages can be overlapping, for example "announcements" is a subclass of "treatments" and "tones" intersects with "treatments" (*ringtone* is a tone, but not

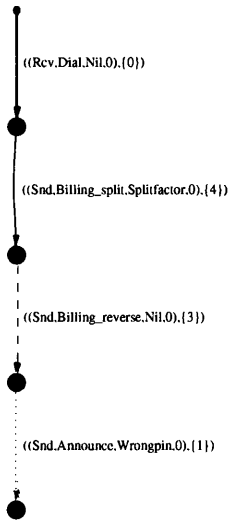

Fig. 6.6: The Constructed Solution Space for Example 6.3.1 after Choosing one Trace

a treatment, whereas *busytone* is both a tone and a treatment and *announce* is a treatment but not a tone).

We wish to express rules that describe sequences of messages. These sequences are required to place an ordering on the occurrence of messages, but also to express the absence of a message. Often we wish to refer to whole classes in a simple way. We have seen such an example before: "a treatment following an onhook message is not useful unless there was an offhook in between".

We use regular expressions to express these rules. Empirical evidence shows that regular expressions were sufficient to express any rule we required. Extra capabilities of context free grammars like counting the occurrence of certain messages was not required in any of the occurring cases. This was to be expected, as messages send to setup, manipulate and tear down calls result in events that must be interpreted by a human user and a telephone system does not usually require the user to count the occurrence of events, i.e. the third ring is equal to the first in its meaning. The used data type for regular expressions is:

$$
\begin{aligned}
\mathit{data}\ \mathit{RegExpr}\ \alpha = & \mathit{MkRE}\ \alpha \\
& \mid \mathit{Concat}\ (\mathit{RegExpr}\ \alpha)\ (\mathit{RegExpr}\ \alpha) \\
& \mid \mathit{Union}\ (\mathit{RegExpr}\ \alpha)\ (\mathit{RegExpr}\ \alpha) \\
& \mid \mathit{Kstar}\ (\mathit{RegExpr}\ \alpha) \\
& \mid \mathit{Plus}\ (\mathit{RegExpr}\ \alpha)
\end{aligned}
$$

Example 6.5.1 Let **treatments** be the class of all announcements and tones (apart from *ringtone*).
Then "a treatment following an onhook message is meaningless unless there was an offhook in between" can be expressed as the following pruning rule (i.e. the rule describes unwanted behaviour) using the textual notation:**onhook.(-offhook)*.treatments**

Having decided to use regular expressions, a good repertoire of matching techniques is available. We can adapt standard algorithms for matching regular expressions on strings to algorithms for matching regular expressions in trees. We describe details of this in the following section.

6.5.1 Implementation

Regular expression matching is performed with the aid of a deterministic finite state automaton (DFA). The DFA is executed on an input string, and a match is achieved when the DFA reaches a final state. We will now consider the construction of the DFA and then show how the matching is performed.

The DFA is constructed from a regular expression in two steps, following the description in [ASU86]. First a non-deterministic finite state automata (NFA) is constructed from a regular expression. The NFA is then converted into a DFA. A Haskell implementation exists by Thompson [Tho01], but this works on strings of characters and therefore was adapted for our purpose (the implementation is not very efficient, but we discuss this later).

In general, a DFA accepting the same language as an NFA is constructed using a subset construction algorithm and thus can be exponentially larger in the number of states. However, "in practice this worst case occurs rarely" [ASU86, p117]. There are other algorithms for the construction of DFAs from regular expressions. For example, McNaughton and Yamada's [MY60] technique can be used to convert a syntax tree into a DFA, however they are of similar runtime complexity.

It is possible to execute the NFA, i.e. determine whether the input is accepted by the NFA, and thus save the extra effort required to generate the DFA. However, simulation of the DFA only depends on the length of the input, i.e. it has complexity $O(n)$ whereas simulating the NFA has complexity $O(nm)$ as it depends on both the length of the input and the length of the regular expression[2]. An optimisation of the runtime of the matching could be achieved by optimising the DFA [ASU86, pp141]. However, we discard such considerations here to maintain a clear implementation.

As the regular expression describing the properties (i.e. unwanted behaviour) usually remains unchanged for longer periods, the construction of the DFA introduces a startup latency which is easily justified by the large number of simulations that is required: whenever an interaction is detected the whole solution space must be searched for any occurrences of matching sequences.

A matching pattern is found in a string when the automata executed on that string has reached an accepting state. It is required to attempt a match from every position of the string, thus for a string of length n, n executions of the automata are required.

Our input is a tree rather than a string, hence we must attempt to match starting from every position of every trace to ensure that all traces of the tree are searched for

[2] Myers [Mye88] shows an $O(\frac{nm}{logn})$ algorithm using the four Russians paradigm.

possible matches (note we match on messages, i.e. on tuples rather than on characters). We will now consider the implementation, showing how the general algorithm for searching for patterns in strings can be adapted to search for patterns in trees. Let *onlymarkers* :: *Ord* $\alpha \Rightarrow$ [*Tree* α] $\rightarrow$ *Bool* be a function returning true if all trees in the list have the value `DelMarker`, let *compact* :: *Ord* $\alpha \Rightarrow$ [*Tree* α] $\rightarrow$ [*Tree* α] be the function that removes any trees with values `DelMarker` or `EmptyTree` from the list (returning a potentially empty list).

```
prunetraverse     :: Ord α ⇒ RegExpr α → Tree (α, Int) → Tree (α, Int)
prunetraverse re t =prunetraverse1 (re2dfa re) t


prunetraverse1                  :: Ord α ⇒ DFA α → Tree (α, Int) → Tree (α, Int)
prunetraverse1 dfa EmptyTree =EmptyTree
prunetraverse1 dfa InsMarker =InsMarker
prunetraverse1 dfa DelMarker =DelMarker
prunetraverse1 dfa (Mk n xs) | onlymarkers pl = DelMarker
                             | otherwise = useDFA2prune dfa (Mk n (compact pl))
  where
    pl = map (prunetraverse1 dfa) xs
```

After the DFA for the given regular expression has been constructed, we traverse through the tree, attempting to match the regular expression from the root. Nested in the traversal new matching attempts are performed on the subtrees – upon finding a match the tree is pruned. Pruning is represented by replacing a tree with the special base case `DelMarker`. Should the pruning result in all subtrees of a node being removed, we continue the pruning by replacing the node with a `DelMarker`. Otherwise the list of subtrees is compacted, meaning that all `DelMarker` entries are removed.

```
useDFA2prune                  ::Ord α ⇒ DFA α → Tree (α, Int) → Tree (α, Int)
useDFA2prune (s, a, tr) input| isMemberSet a s = DelMarker
                             | otherwise = useDFA2prune1 a tr s input
```

After testing whether we have already reached an accepting state we recurse through the tree. Note, here we actually progress through the DFA as well as the tree, rather than just shifting the starting point of the matching attempt. Clearly an `EmptyTree` or a `DelMarker` does not match the regular expression, so we cannot prune such a branch. In the case that a node with subtrees is found, the subtrees are pruned. Let *goto s a* be a function returning the target state reached from the current position *s* by following the transition labelled *a*. After applying the function two choices are possible:

- An accepting state is reached, in which case the respective subtree must be pruned.

- *goto s a* returned −99, i.e. we have found a transition leading to an error state – meaning that we have not found a match.

Again (as with the traversal to shift the start of the matching attempt), if all subtrees have been pruned the parent is removed as well, otherwise `DelMarker` trees are removed from the list of subtrees.

$$
\begin{aligned}
&\mathit{useDFA2prune1} && :: \mathit{Ord}\ \alpha \Rightarrow \mathit{Set}\ (\mathit{Set}\ \mathit{Int}) \rightarrow \mathit{TTBL}\ \alpha \rightarrow \mathit{Set}\ \mathit{Int} \\
&&& \qquad \rightarrow \mathit{Tree}\ (\alpha,\ \mathit{Int}) \rightarrow \mathit{Tree}\ (\alpha,\ \mathit{Int}) \\
&\mathit{useDFA2prune1}\ a\ tr\ s\ \mathit{EmptyTree} &&= \mathit{EmptyTree} \\
&\mathit{useDFA2prune1}\ a\ tr\ s\ \mathit{InsMarker} &&= \mathit{InsMarker} \\
&\mathit{useDFA2prune1}\ a\ tr\ s\ \mathit{DelMarker} &&= \mathit{DelMarker} \\
&\mathit{useDFA2prune1}\ a\ tr\ s\ (\mathit{Mk}\ n\ xs) && \mid \mathit{isMemberSet}\ a\ gto\ =\ \mathit{DelMarker} \\
&&& \mid \mathit{gto}\ ==\ -99\ =\ (\mathit{Mk}\ n\ xs) \\
&&& \mid \mathit{onlymarkers}\ pl\ =\ \mathit{DelMarker} \\
&&& \mid \mathit{otherwise}\ =\ \mathit{Mk}\ n\ (\mathit{compact}\ pl)
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{gto}\ &=\ \mathit{goto}\ s\ tr\ (\mathit{fst}\ n) \\
\mathit{pl}\ &=\ \mathit{map}\ (\mathit{useDFA2prune1}\ a\ tr\ \mathit{gto})\ xs
\end{aligned}
$$

In the following example we apply pruning to the example tree Fig. 6.1. Clearly the given billing messages are contradictory, as with one the caller should not pay anything, with the other the caller carries a share of the call costs. Hence, it is not meaningful to commit to a branch with both those messages present.

Let *non-bsbr* be the class of messages containing all messages apart from *billing_split* and *billing_reverse*. The regular expression describing the unwanted solutions is as follows:
(billing_split.(non-bsbr).billing_reverse)—(billing_reverse.(non-bsbr)*.billing_split).*

The tree shown in figure 6.7 is obtained from example 6.3.1 using $\mathit{fm}\ fs\ m\ \mathit{rules} = \mathit{prune}\ (\mathit{construct}\ fs\ m)\ \mathit{rules}$ and the following instantiation of *rules*:

$$
\begin{aligned}
&(\mathit{Union}\ (\mathit{Concat}\ (\mathit{Concat}\ (\mathit{MkRE}\ (\mathit{Snd}, \mathit{Billing\_split}, \mathit{Splitfactor}, 0)) \\
&\qquad\qquad (\mathit{Kstar}\ (\mathit{Union}\ (\mathit{MkRE}\ (\mathit{Rcv}, \mathit{Dial}, \mathit{Nil}, 0)) \\
&\qquad\qquad\qquad (\mathit{MkRE}\ (\mathit{Snd}, \mathit{Announce}, \mathit{Wrongpin}, 0))))) \\
&\qquad (\mathit{MkRE}\ (\mathit{Snd}, \mathit{Billing\_reverse}, \mathit{Nil}, 0))) \\
&\quad (\mathit{Concat}\ (\mathit{Concat}\ (\mathit{MkRE}\ (\mathit{Snd}, \mathit{Billing\_reverse}, \mathit{Nil}, 0)) \\
&\qquad\qquad (\mathit{Kstar}\ (\mathit{Union}\ (\mathit{MkRE}\ (\mathit{Rcv}, \mathit{Dial}, \mathit{Nil}, 0)) \\
&\qquad\qquad\qquad (\mathit{MkRE}\ (\mathit{Snd}, \mathit{Announce}, \mathit{Wrongpin}, 0))))) \\
&\qquad (\mathit{MkRE}\ (\mathit{Snd}, \mathit{Billing\_split}, \mathit{Splitfactor}, 0))))
\end{aligned}
$$

Note that in this example *non-bsbr* is represented by *Dial|Announce*, so only these messages together with *Billing_Split* and *Billing_Reverse* (as opposed to all possible messages) occur in the example. This is to simplify the example.

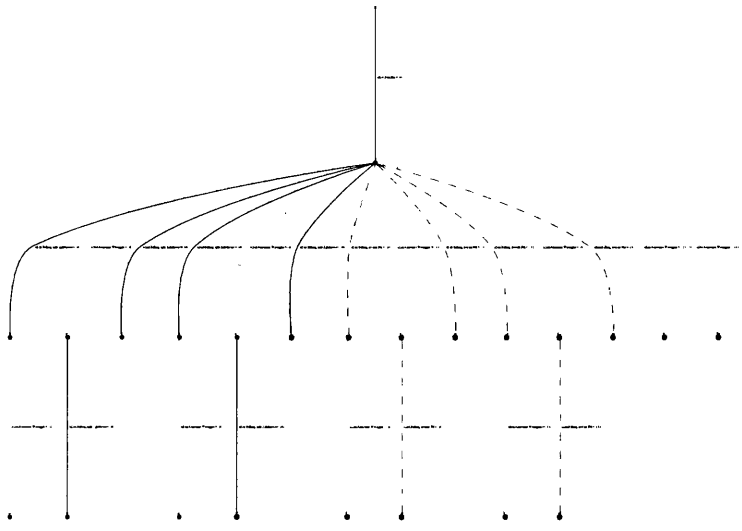

Fig. 6.7: The Constructed Solution Space for Example 6.3.1 after Pruning

6.6 Application Order and Necessity of Rules

We have discussed two categories of rules, message dependent rules and message independent rules. The latter consist of more than one rule. An obvious question to ask at this point is what is an application order of rules? This only arises as a question when the application order of rules matters, which we consider from a functional and a performance aspect.

We have devised four message independent rules: removing duplicates, satisfying features, priorities and selecting one resolution. Further we have one message dependent rule, pruning using pattern matching. We have already identified that most of the message independent rules are unsafe, the exception being the removal of duplicates. Clearly, this suggests that application order is critical.

Recall that extraction rules (i.e. message independent ones) remove **resolutions**, whereas pruning removes unacceptable **solutions**. Removing unacceptable solutions is certainly desired, thus we can consider pruning safe.

The suggested application order is:

1. extract duplicates
2. prune
3. extract traces satisfying most features and then apply extraction by priorities (*or* vice versa)
4. select one resolution

We can also distinguish the presented rules by their necessity. Clearly pruning is essential, as it removes undesired behaviour. Selecting one branch is also required, ensuring that indeed only one resolution remains. The remaining rules influence performance (removing duplicates) or the quality of solutions (priorities, satisfaction of most features). Having established priorities allows for a more fine tuned resolution, as user preferences can be reflected in the priorities.

In conclusion, application order is relevant as some rules are safe and others are unsafe. Moreover, different application orders of unsafe rules result in solutions of different quality. We also note that some rules need not be applied at all, thus changing the quality of the solution.

6.7 On-the-fly Pruning

So far we have presented construction and pruning independent of each other, indeed we have assumed that pruning is applied to a fully constructed solution space. This approach is referred to as "construct-prune".

A major drawback of this approach is the complexity: a large number of branches must be constructed that will be discarded. For a runtime approach, complexity is a very important issue. We therefore propose an "on-the-fly" approach.

Basically, on-the-fly resolution works by trying to apply pruning to the current solution under construction. As soon as the behaviour is identified as undesired, the construction of the current solution is aborted. The part constructed since the last choice is then removed and construction of another solution is attempted. This will greatly reduce the complexity if many features are involved; bad solutions can be identified early in the construction.

Clearly, both construct-prune and on-the-fly should result in the same resolution being found. Note that message independent rules, i.e. extractions, can **not** be embedded in the "on-the-fly" process but still have to be applied to the resulting solution space – until the complete solution space has been constructed, we do not know that two subtrees are equal. In the same way, before the construction is complete we do not know which solution satisfies the largest number of features.

The on-the-fly mechanism requires some minor changes to the implementation. Notably a new insertion procedure for the solution tree is required, which does not insert unwanted branches and reports the success of an insertion. The feature manager is required to react to the feedback from an attempted insertion and proceed with its abort-commit mechanism accordingly.

We now discuss the details of the required changes and provide an example.

6.7.1 Implementation and Example

The on-the-fly algorithm uses a tree that in addition to the information in an `Augtree` stores a set of states of a DFA at each node. We refer to this data structure as `OTFTree`. Once the on-the-fly algorithm terminates the tree is converted into an `Augtree` (only messages and a set of feature numbers stored at each nodes) as used in the Construct-Prune approach. This conversion is trivial – the additional information on DFA states in each node is removed.

Insertion of elements is handled by:

$$\begin{aligned} \mathit{insertOTFTree} :: \mathit{Ord}\ \alpha \Rightarrow\ & \mathit{OTFtree}\ \alpha \rightarrow (\alpha,\ \mathit{Set\ Int}) \rightarrow \mathit{DFA}\ \alpha \\ & \rightarrow (\mathit{Bool},\ \mathit{OTFtree}\ \alpha) \end{aligned}$$

The insertion in the `Otftree` is more complex than insertion in the `Augtree`. Again an `InsMarker` is used to indicate the next insertion position. The element to be inserted is, as before, a pair consisting of a message and a set of feature numbers. Insertion returns a pair (*Bool*, *Otftree* α), where the first element indicates the success of the insertion and the second contains the new tree. If the insertion is successful, the new element together with a set of DFA states will have been inserted. Should the insertion fail, a `DelMarker` is inserted.

This leaves two open questions: *when does insertion fail?* and *why do we need the DFA states?* Recall that the DFA is the recogniser for patterns in the pruning rules. A match means that we have found a sequence of messages that is undesired; we wish to prune this branch. Clearly, this answers the first question – insertion fails when we have found a sequence matching the rules. The DFA states stored represent all possible states the DFA can be in at this point. Initially the DFA is in its start state, we insert the root node after which the DFA can still be in its start state (we could start a new search from here onwards). It could also be in the state reached by following the transition labelled with the trigger event. With each inserted node the set of DFA states increases provided the DFA can reach a new state from any of the current states.

Obviously a test for failure of insertion must be incorporated in the feedback process of the feature manager – if insertion fails we want to rollback immediately. Recall that feedback distinguishes 4 cases, but only in one case can insertions occur. That is when we still have messages in the response queues waiting for feedback and we have not explored all possibilities. Hence we are only required to adapt this case. We actually split this case by distinguishing success of insertion as an extra condition. If the insertion is successful, we proceed in the same fashion as before, otherwise we simply rollback. The rollback on failure of insertion is equivalent to a rollback when we cannot explore a branch any further.

Some minor changes have been implemented to allow as much code reuse as possible. The feature manager can be customised by a flag depending on whether the on-the-fly

or the construct-prune approach is required. The *construct*, *construct*1, *feedbackctrl*, *feedbackctrl*1 and *feedback* functions have been duplicated and adapted to handle the different tree datatype. The *feedback* also handles the new insertion routine. Note that *construct* converts the `OTFTtree` into an `Augtree` for further processing by extraction.

The respective function declarations are:

otfconstruct :: [*CState*] → *Message* → *RegExpr Message* → *Augtree Message*
*otfconstruct*1 :: [*CState*] → *Message* → *DFA Message* → *OTFtree Message*
otffeedbackctrl :: ([*Queue Message*], [*CState*]) → *OTFtree Message*
→ *DFA Message* → *OTFtree Message*
*otffeedbackctrl*1 :: [[*Bool*]] → [*Queue Message*] → [*CState*] → *OTFtree Message*
→ *DFA Message* → *OTFtree Message*
otffeedback :: [*Bool*] → [*Queue Message*] → [*CState*] → (*OTFtree Message*)
→ *Int* → *StateStack* → *DFA Message* → (*OTFtree Message*, [*CState*])

Example 6.7.1 Applying the on-the-fly approach to Example 6.3.1 yields the same solution space as shown in Fig. 6.7.

Example 6.7.1 shows that the expected result was obtained. This was obtained using the same time but slightly less memory (3,789,920 bytes as opposed to 3,877,672 bytes)[3] than the construct-prune method. This does not look very promising but, we need to consider that in the given example most traces only match once the traces are completely constructed. Consequently, not much exploration work can be saved (and in fact because there might be duplicate branches the matching has to be done more often). We have seen examples where the on-the-fly method outperformed the construct-prune method roughly 2:35 (7 features, shown in Fig. 7.3). As we show when analysing the complexity in section 7.3 the worst case behaviour of the on-the-fly approach is indeed similar to that of the construct-prune approach, but in larger scenarios with deep trees a significant saving can be made by early pruning.

6.8 Summary

We have identified a number of rules which allow us to obtain resolutions by applying respective operations to the solution space. The rules fall into two categories: message dependent and message independent. The application order of rules was discussed. A more efficient on-the-fly approach has been developed thus tightly integrating construction and pruning.

[3] Time and memory measurements were performed using `ghcprof`

Chapter 7

Evaluation

7.1 Introduction

In this chapter we consider the correctness and complexity of our approach. We discuss why our approach is transactional, but does not coincide with the classical model. In addition we apply our method to two distinct feature sets: the features used by Marples [Mar00] and the running example from Chapter 3.

Having shown that the approach is correct we consider whether it is indeed appropriate and suitable to solve the posed problem. We clarify the role of the semantics of messages.

We revisit the idea of a hybrid approach as discussed in Chapter 4.

7.2 Correctness

In order to show that the construct-prune and the on-the-fly approach proposed in the previous chapters actually deliver the correct results we need to analyse their behaviour. We identify three theorems that define what we mean by *correctness* (of construction, pruning and the on-the-fly approach respectively).

Theorem 1 (Correctness of Construction) The solution space is constructed correctly iff

1. all possible interleavings are inserted and
2. every trace in the solution space can be generated by the feature automata and
3. the construction terminates

Theorem 2 (Correctness of Pruning) Pruning is correct iff

1. no trace in the resulting resolution space violates the properties defined by the rules and
2. every trace from the solution space not violating rules is part of the resolution space and

3. the pruning terminates

Theorem 3 (Correctness of the on-the-fly Approach) The on-the-fly approach is correct iff

1. every trace in the resolution space can be simulated by the feature automata and
2. no trace in the resulting resolution space violates the properties defined by the rules and
3. every possible interleaving not violating rules is part of the resolution space and
4. the process terminates

7.2.1 Proving Correctness

We have developed our implementation in Haskell, under the assumption that it is possible to reason about program behaviour. The Haskell implementation provides a basis for a formal proof, as the implementation is essentially in the form of left-right rewrite rules. However, a proof requires a further formalising of the approach, depending on the automated technique used. This formalisation is a large task and is not mathematically interesting.

Furthermore, as the whole system depends on the features, it would be desirable to reason about arbitrary features. Clearly we can only reason about the implemented features – it is not possible to reason about features in general, as no coherent structure of the same exists. Thus, the proof would be based on case analysis, where splits are made dependent on the features under consideration. We believe that in the given context this type of reasoning is undesirable as the detail distracts from the concept.

We therefore reason on a more conceptual and less formal level. In the following we aim to provide insight into why the presented theorems hold without losing sight of the intuition behind the approach.

7.2.2 Correctness of Construction

1. Insertion of Interleavings

The construction of the solution space ensures that if there are no responses to a trigger only the trigger will be inserted into the solution space. If only one feature responds to a trigger and there is no further response triggered by the feedback process, only the single response is inserted. In both these cases no interleaving occurs, so there is only one possible trace (which is inserted); hence the insertion is trivially correct.

Two cases remain to be considered: more than one feature responds to a trigger and further responses are triggered by fed back messages. We will discuss these in turn.

If more than one feature responds to the initial trigger, each feature's responses are placed in a separate queue. Crucially, having separate queues allows one to maintain the relative order of the responses. The feedback process has been described in detail in section 5.9.1. In effect, the feedback mechanism ensures that all possible permutations of the response messages (subject to the relative order being maintained) are tried, and each is inserted into the tree. This essentially is the generation of all possible overlapping interleavings as described in section 5.4.

The process is the same when fed back messages trigger further responses. The new responses are simply added to the end of the respective queues. Clearly these further responses are only interleaved with parts of the traces that still need to be constructed, as they should never occur before the respective trigger event. Adding new responses to the end of the queues maintains the relative order.

We can conclude that all interleavings are indeed constructed.

2. Traces and Feature Automata

Features are described by finite state automata, as shown in Appendix A. By the feature automata being able to simulate a trace we mean that when considering a trace, there is at least one automaton that can make a move for each element in the trace. Recall that transitions are labelled with an input/output pair.

Considering examples, we can distinguish three cases:

- The trace is of the form $t.ms$, where t is the trigger event and ms are the responses of one feature (i.e. there is a feature automaton with a transition t/ms.
- The trace is of the form $t.m_1.m_2...m_n$ where t is the trigger event and the trigger event and $m_1, m_2...m_n$ is an interleaving of responses from several features (i.e. there are several feature automata with transition labelled by t/m with the (between features possibly different) m being composed of any of $m_1, m_2...m_n$ maintaining the relative order and ensuring that all m_i occur in at least one transition).
- The trace is of the form $t.m_1.m_2...m_k...m_n$, where again t is the trigger event and the m_i are responses from one or more features. Note that m_k is a response of one feature that (upon feedback) triggered a further response. In this case there is at least one feature automaton with a transition t/m with m containing m_k and one with a transition labelled by m_k/M where M is composed of any of $m_{k+1}...m_n$ again maintaining the relative order.

3. Termination

Constructing the solution space is general recursive (as opposed to primitive recursive), as the recursion occurs over a number of arguments that might increase or decrease depending on the feature behaviour. As an example, the message queues can shorten when no features respond but might also lengthen when features respond. Rollback reduces the size of the state stack, but new transactions increase the size. So in general, construction might not terminate. However, as the construction is bounded by a *maximum depth* and the constructed tree has only finite width, construction indeed terminates as explained below:

Looping behaviour of features can lead to infinite traces. Recall that one of the drawbacks of the initial specification was the restriction to finite traces. We can distinguish two kinds of loops: those where one feature produces looping behaviour and those where loops occur because of the mutual triggering of several features.

Clearly the case that one feature produces looping behaviour as result of a single trigger event should not occur. Although features should be able to loop to an earlier state (e.g. call waiting which allows toggling between two calls) features should not respond with an event that re-triggers them. The latter would cause non-termination of a single feature and eventually break the system. Thus we can conclude that this behaviour could only arise as the result of an incorrect implementation of the given feature.

Assuming correct implementation of all features, looping behaviour might still occur between a number of features. The most obvious example being a call forwarding loop: Two users subscribe to *call forwarding unconditional*, where user A's forwarding is to B and B's forwarding is set to A. Upon User A receiving a call, the call will be forwarded to B to be forwarded to A, and so on. This behaviour could potentially prevent the termination of the construction algorithm.

To show that the algorithm terminates we need to show that the constructed tree is indeed of finite depth and breadth.

By allowing the constructed tree to only extend to a specified **maximum depth**, we can ensure that the tree will have finite depth. There are three cases, shown in Fig. 7.1: a) All traces terminate before the maximum depth is reached, b) some traces terminate after the maximum depth is reached and c) all traces terminate after the maximum depth is reached.

Case (c) should never occur by judicious choice of the maximum depth. Recall that the solution space contains traces produced by the execution of a single feature, in addition to all the possible combinations. Thus, we have a minimum maximum: the maximum length of a features trace.

The maximum depth can be chosen arbitrarily large, however a sensible bound should take into account that the depth should be reachable in short period of time. We assumed that a feature can safely operate within depth 10 (that is a trigger produces

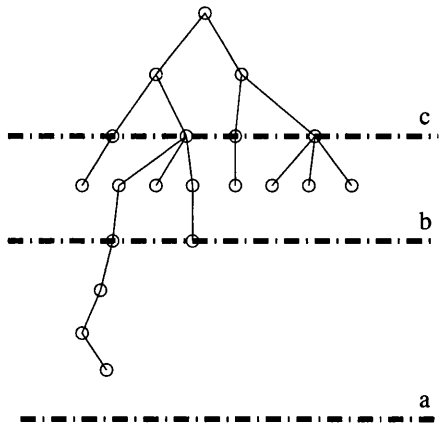

Fig. 7.1: Solution Space and Maximum Depth

no more than 10 response messages from one feature). In fact for the given features the maximum is 8 for group ringing, mostly features respond with 1 to 3 messages. A maximum size of 50 was chosen in the model to provide enough margin for interleaved traces.

The mutual triggering leading to looping behaviour discussed previously could result in case (b), i.e. that some traces are deeper than the given maximum depth, and forms the main reason for the introduction of the boundedness in the algorithm. The described scenario forms an infinite loop which maps onto an infinite trace – this case needs to be dealt with to prevent non-termination of the algorithm.

Some traces might be longer than the maximum depth without resulting from infinite looping behaviour. Here it could be argued that potentially good behaviour might be presented by the trace formed as interleaving of a number of features. However, recalling that time is at a premium in a runtime approach, it is justifiable that constructions continuing for too long are terminated. Should this case occur too frequently, the chosen maximum depth might be non-optimal and can be increased accordingly.

Case (a) represents the setting where no loops occurred and all the interleavings could be constructed within the provided depth bound.

Finite breadth is ensured as at every node in the solution space it is known how many children there will be. In addition the number of children is always finite, because it is equal to the number of non-empty response queues. As there is one queue for each feature the number of children is at most equal to the number of features.

New insertions only occur when we have tried to construct fewer subtrees than the current node can have. Each subtree starts with a message taken from the front of a queue. Placing an order on the response queues allows us to order the subtrees thus providing a means of guaranteeing that no subtree is generated twice.

Note, that the construction process is similar to a bounded depth first search. We have shown that the depth and breadth of the tree are finite and also that no insertion is attempted twice. From this we can conclude that construction terminates.

7.2.3 Correctness of Pruning

Under-Pruning

The implementation of the pruning process was described in section 6.5. We must prove that all traces which match the pattern describing bad behaviour are removed.

The pattern, i.e. the message sequences describing bad behaviour, is provided as a regular expression. The expression is converted into the corresponding automaton which is used by the matching algorithm – this is a standard technique.

The matching has to be performed on the solution space which has a tree structure. The tree is searched recursively, that is after attempting to find a match at the root node, we move on to the child nodes. At each child node we continue the match started earlier but also start a new matching process; in some sense one could say that a number of match processes are run concurrently.

When a match is found, the node is replaced with a special leaf node, `DelMarker`. If the node is a leaf node, then the current trace needs to be removed back to the last "decision point". If we find the match in an internal node the trace since the last decision point needs to be removed regardless of any subsequent messages. As an example, consider a user going onhook and then receiving an announcement. The user has received an announcement after going onhook, and so regardless of the events occurring afterwards, this trace must be removed.

Once a `DelMarker` is inserted in the tree, we do not follow this path any further. Equally, not finding a match on a trace and reaching a leaf node means we do not follow the path any further. In both cases we backtrack to the parent node. At the parent node we consider whether all children are marked for deletion, if so the parent is marked. If not, all children marked for deletion are simply removed. This process continues up through the tree until the root node is reached and we cannot backtrack any further.

Figure 7.2 shows an example solution space and the resulting resolution space on pruning assuming the pattern describing unwanted behaviour to be *abaa*.

The remaining tree does not contain any traces that match the given pattern, hence only traces that are not violating the described property remain after pruning, hence under-pruning does not occur.

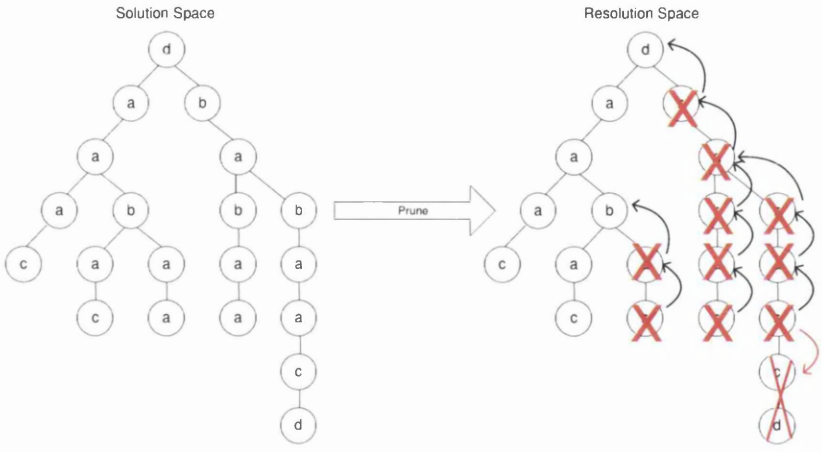

Fig. 7.2: Pruning – an Example

Over-Pruning

In the previous section we have shown that the deletion of nodes is propagated backwards through the tree. One could argue that we could delete too many nodes by doing this. We investigate now why we do indeed delete the right number of nodes.

We consider three cases: the removed part of a trace starts with the first element in the pattern and then includes the complete pattern; the removed part of a trace is only a suffix of the pattern and the removed trace contains events earlier than the pattern started (Cases two and three can be seen in Fig. 7.2). We will justify all three cases now.

If the removed trace starts with the first element of the pattern, we have removed obstructing behaviour. Possibly behaviour following the obstructing part was also removed, but this has been justified in the previous section. Hence in this case no over-pruning can occur.

The case that only a suffix of the pattern is removed seems more likely to cause underdeletion, as we might not have removed all obstructing behaviour.

On the other hand, if all subtrees of a node have been deleted, how do we justify propagation of the deletion back towards the root, and potentially before the first message of the matching pattern occurred? This case occurred in the right subtree of the example above.

Assuming that we have deleted all subtrees of a node, consider the meaning of the trace from the root to this node. The behaviour is only partial w.r.t. the expected behaviour of the features. A trace up to a certain node expresses messages taken from a number of features that have been interleaved. By this point however the behaviour of all features from which messages have been taken is incomplete and the trace should be removed. For example, we could find situations were a user is connected but billing is not initiated at all, because the billing message occurred later in the removed part.

Assume a trace with 4 features, where two produce the conflicting behaviour and this occurs after the other two have completely exhibited their behaviour. The backward pruning will remove the good behaviour of the other two features as well, but there will also be traces in the solution space that are generated by just the non-obstructing features being active – hence no possible resolution has been removed. In general, when some features have exhibited their complete behaviour before the point at which the sequence is removed, backwards pruning will only remove behaviour that exists in the solution space at a different place.

In the extreme, backward propagation could lead to the removal of the root node and hence all solutions would be deleted – clearly this would be considered as overpruning. The case that all subtrees of the root node have been deleted will not occur, as some subtrees represent behaviour of individual features which we always deem to be correct.

In section 7.2.2 we have shown that all possible traces are inserted. In this section we saw that the deletion process only removes those traces that violate the given properties. Hence all traces not violating the properties are part of the resulting solution space.

Termination

Pruning consists of

1. construction of the DFA from a regular expression
2. traversing the tree to start the matching from every position (*prunetraverse*1)
3. matching the regular expression to a path (*useDFA2prune*)

The construction of the DFA from the regular expression uses a standard algorithm, details of which were discussed in chapter 6. The algorithm is known to terminate.

However, the traversal of the tree combined with the matching is non-standard, hence it should be shown that both parts do indeed terminate.

Both functions are defined by primitive recursion over trees. Trees are ordered by a natural well-founded ordering: any tree is larger than its subtree.

In the case of the tree data type used in our model:

- `InsMarker`, `DelMarker`, `EmptyTree` are trees
- `(Mk n` $t_1 \ldots t_k$`)` is a tree
- nothing else is a tree

We define the ordering $>$ on trees according to the size of the tree.

$$InsMarker = DelMarker \qquad = EmptyTree$$
$$(MkTree\ n\ t_1...t_k) > InsMarker$$
$$\forall(i|1 \leq i \leq k).(MkTree\ n\ t_1...t_k) > t_i$$

All that remains to be shown is that the functions *prunetraverse*1 and *useDFA2prune* indeed reduce the size of the argument in the recursive case and that they consider the base case. Both functions define return values for the base cases of the tree data type and the recursive case follows the general format $f(Mk\ n\ t_1...t_k) = f(t_1)...f(t_k)$. Our ordering on trees confirms the required reduction of argument size, as per definition $\forall(i|1 \leq i \leq k).(Mk\ n\ t_1...t_k) > t_i$.

7.2.4 Correctness of the On-the-fly Approach

Traces and Feature Automata

By similar reasoning we argue that all traces that are part of the solution space can indeed be simulated by feature automata. The only difference in this case is that the solution space constructed by the on-the-fly method does not contain any traces that violate the pruning rules. The on-the-fly construction simply inserts fewer traces into the tree. The resulting solution space forms a subset of the solution space generated by the construct-prune approach (immediately after construction). Hence the traces in the current solution space would also exist in the full (i.e. un-pruned) solution space as constructed earlier. As all traces from the full solution space can be simulated by feature automata, all traces constructed by the on-the-fly method can be simulated.

Traces and Rules

The data structure used to store the solution space is a slightly extended version of that used in the construct-prune approach: in addition to the information stored in nodes in the construct prune approach, we store a list of states of the matching automata. This state list contains all possible states of the matching automata, assuming it had been restarted and continued in any of the previous nodes along the current trace.

When a new insertion is performed, it is first checked whether the new element would cause the matching automata to reach an accepting state from any of the possible states. If this is the case, we insert the deletion marker and discontinue construction.

This leads to a solution space where we never insert any trace that violates the rules, which is exactly what we require.

Completeness of Solution Space

Inserting new nodes into the tree is performed in exactly the same way as in the construct-prune approach, provided no rule is matched upon insertion. When a rule is matched no insertion of the new message occurs, we rather insert the special leaf `DelMarker` as described before.

Upon rolling back we apply the same backwards propagation of the `DelMarker` as discussed in the pruning method.

The arguments for the construction and for pruning allow us to conclude that indeed all traces not violating the specified properties are inserted into the solution space, thus it is complete, i.e. no possible resolutions are missing.

Termination

The same argument as that used for construction can be applied here. The only difference is that before a message is inserted into the tree a check is carried out as to whether this message results in a match w.r.t. a pruning rule. The check for match is simply a call to a function of the matching DFA to determine whether the message will result in the automaton reaching an accepting state.

In case a match occurs, the trace under construction is aborted and a new trace is attempted. Note that again the depth is finite, as limited by the given bound. The width does not increase, as matching a rule only leads to a non-insertion of the message. Again no traces are constructed twice, as the same construction mechanism as before is used.

We can conclude that the on-the-fly approach terminates.

7.3 Analysis of Complexity

We consider the complexity of the construction, the pruning and the on-the-fly approach in turn.

Construction. The complexity of construction is best measured w.r.t. the size of the constructed solution space. Construction consists of issuing a message, collecting the responses, inserting one message in the solution space and issuing the next message. Occasionally rollback occurs.

Sending a message to the features and collecting the responses is of complexity $O(1)$, i.e. it takes constant time. Hence we can ignore this. The other two factors depend on the size of the solution space.

Insertion involves a search through the solution space to find the next insertion point. We have stated that insertions always occur in the leftmost branch of the tree. Hence,

finding the insertion point means that, in the worst case, we need to search to the end of the leftmost branch.

Rollback means restoring some features to an earlier state (which can be assumed to be performed in constant time) and also moving the insertion marker. The latter is in complexity similar to insertion, the marker must be found after which moving is straight forward. As both insertion and moving the insertion marker depend on the length of the leftmost branch we should consider how long this will be. The length of a branch depends on the number of features involved. With just one feature, a branch will be as long as the sequence of responses to the trigger event. With a number of features, the length of the branch is simply the sum of the length of the responses of the features. Hence, if we have many interacting features the branches will be longer and furthermore if the responses of each feature grow, so too will the length. To give an upper bound, the longest branch will be at most $O(max(nm, maxdepth))$, where n is the number of features and m is the length of the longest response and $maxdepth$ is the maximal depth.

The number of branches inserted into the tree depends on the number of features and the length of their responses. In fact, the size of the solution space is exponential in the number of features and the length of the responses. However, this is a worst case bound, which in practice rarely occurs. For the solution space to reach this worst case, all features present would be required to be triggered and add to the responses.

Practical experiments showed that the considered solution spaces have been constructed relatively quickly. Figure 7.3 shows the time required for construction as well as the size of example solution spaces from the running example. The results were measured by the Haskell profiler `ghcprof` and all experiments were conducted on a P3-450 running the Linux Operating System. The smallest time-unit reported by the profiler is 0.02 seconds, hence results of 0 mean that no more than 0.02 seconds have elapsed. The indicated times exclude the time required to print the structure to the screen, as this is not part of the real construction effort. Note that the representation of the runtimes uses two different scales on the y-axis to show small (i.e. $< 15secs$) values more clearly.

Pruning complexity is a different issue. Here we can distinguish the construction of the DFA (which has been discussed earlier, it only adds startup latency) and the actual pruning. Pruning depends on the size of the solution space. Recall that we need to attempt to reach an accepting state of the DFA following transactions labelled with the same messages as occur in the tree. This has to be repeated at every node in the tree.

As at every point in the DFA only one transition with a given label is possible, the complexity arises from the number of alternative transitions possible. Starting from any point in the tree and traversing the child trees, a worst case is given by a nested traversal of the tree. However, the inner traversal has a reduced space to search; only the nodes in the subtree must be considered. Furthermore, once a match has been found the search is stopped, as the subtree is immediately removed.

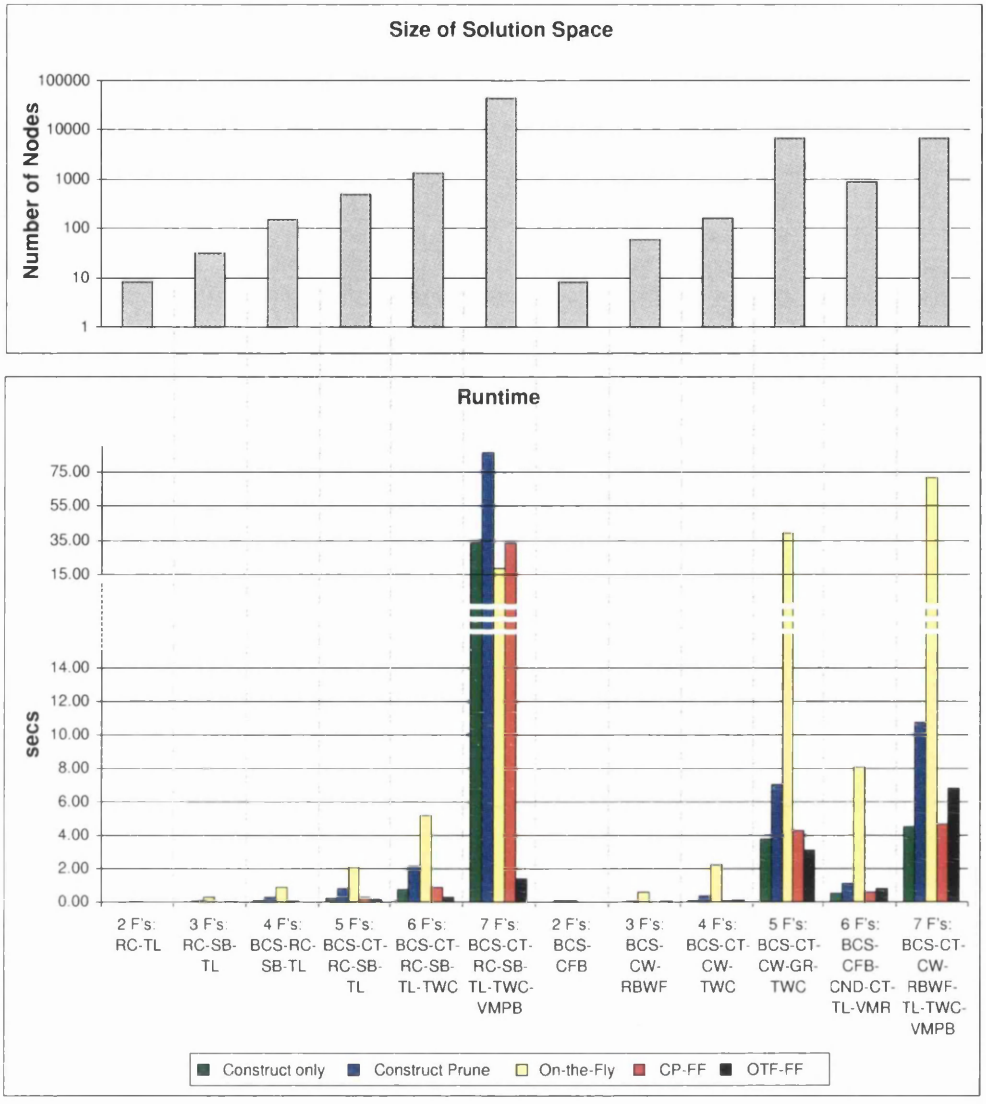

Fig. 7.3: Example Runtimes (Empirical Results)

In practice, runtime seems rather long, as the results in Fig. 7.3 (Construct-Prune) show. Note that the time required to construct the DFA has been extracted from the shown figures to enable a better comparison to be made.

Further analysis indicates that one particular function is the cause of this disappointing runtime. This function is *findfinal* which determines the state we can move to in the DFA given a starting state and a label. Figure 7.3 (CP-FF) shows the results when the runtime for *findfinal* is ignored, which seem much more reasonable. In fact, they are more realistic, as an implementation for a switching system would not use a linear list to store the transition table but rather make use of a hash table. This would provide constant time access, rather than $O(n)$ worstcase complexity (with n being the length of the list) which occurs in our implementation.

On-the-fly. To avoid a large number of solutions being generated just to be removed we proposed the on-the-fly approach. Here pruning occurs during construction, thus one would assume that a smaller solution space is constructed and hence searches are performed faster.

The on-the-fly method requires fewer traversals of the tree, as the pruning occurs during insertion and the relevant information about the potential start points for matching must be stored in the tree. Clearly this increases the footprint of the tree structure.

We note that pruning after the construction is applied to a solution space where duplicates have been removed. Thus the searched space is potentially smaller. The on-the-fly method does not allow for the removal of duplicates before pruning (we only know what a duplicate is when the whole tree has been constructed). Thus, in the worst case, the space searched by the pruning algorithm during the on-the-fly approach can be larger than in the construct-prune approach. However, this worst case occurs only if no pruning is possible or if for every pruning attempt a match can only be achieved when a leaf node is reached, and duplicates do indeed exist in the specific solution space. In general, we are often able to prune early in the construction. This especially applies to long branches, and thus we can make a significant saving. Again, *findfinal* has a significant impact on the runtime, so Fig. 7.3 shows both results with and without the time used by *findfinal* (On-the-Fly and OTF-FF respectively).

Overall, the complexity of the two presented approaches has a startup latency for the construction of the DFA and then during runtime a theoretical worst case of a multiple of a exponential complexity (the solution space is bounded in depth and the width depends on the possible combinations of features), However, this theoretical worst case was never reached in the practical trials, thus making the approach tractable.

Scalability

Scalability is an issue that must be considered, especially in the context of an increasing number of features available to each user. There are two other aspects to scalability; the number of users that are served by one exchange and the number of exchanges. Here we consider primarily the scalability with respect to a growing number of features, as this is the dimension that is motivated in the context of the feature interaction problem. However, we also comment on the other two dimensions.

Considering additional features, we can have two possible outcomes: they do not interact with any of the existing ones or they interact.

A non interacting feature, i.e. one that does not respond to the given trigger or and fed back messages, introduces duplicate traces as explained earlier. That is, it leads to a larger breadth of the solution space, but it will not influence the depth of the solution space. If the new feature does indeed interact with existing features then, in addition to adding more possibilities it also adds to the length of individual solutions. In this

situation the on-the-fly approach facilitates scalability: if the trace must be pruned then chances are that pruning can occur early in the construction, thus ensuring that the full length of the trace need not be explored.

Scalability with respect to a growing number of exchanges is of little relevance, as current communications are between only a few (normally two) users. As we do not exchange information beyond that transmitted in normal communications mechanisms between exchanges, this dimension is not relevant for our approach. However, scalability with respect to growing numbers of users on a single switch is relevant, as all users might be simultaneously involved in calls. In this case a large number of solution spaces has to be generated and as we have seen this can require a significant amount of resources. However, only practical trials on an operational exchange can provide details here. This issue is not only specific to interaction handling, as simply allowing more features being enabled on a switch increases the amount of resources required.

Overall, when we compare the results obtained for construction only with those that also include resolving (either by the construct-prune or the on-the-fly method) we can see that resolution introduces only a minimal overhead. Marples's construction method is essentially identical to ours, and he accepted his runtimes as suitable (at least he does not argue to the contrary). We conclude that our runtimes are like Marples's, and using the on-the-fly method can even undercut his.

7.4 Transactional Approach

We refer to our approach as transactional. This is partly historical, Marples used this term for his approach which we extended. However, the approach makes use of ideas from transactional consistency and error recovery approaches for distributed systems. We will now investigate how close the link is.

Traditionally transactional approaches in distributed systems have been developed to handle error recovery and maintain consistency. There is a significant body of work in this context. The main concerns are to provide a well defined system state after a failure, to delimit the loss of data and to maintain consistency of data. Whilst this is clearly useful in say a banking system, we are not concerned with failure here.

In the context of transactional approaches, four properties are considered: atomicity, consistency, isolation and durability. They are often referred to as ACID properties [Wei89]. The granularity of a transaction is provided by atomicity: a transaction is indivisible. In the case of failure, this means that a transaction has either been executed completely or not at all. Commit is usually used to confirm the complete execution, abort to return to an earlier checkpoint. Consistency means that each transaction, when completely executed on its own preserves the invariants of the system (i.e. transition does not introduce data inconsistencies). Isolation (or serializability) relates to a group of transitions: if they are executed in parallel they

are indistinguishable from being executed in series in any order. Durability means that the effects of committed transactions survive future failure.

As we are not concerned with failure here, durability is not an issue. During the construction of the solution space, we generate checkpoints, to which we return to explore further possibilities. However, the concept of failure is irrelevant.

More interestingly, we have spoken about transactions and we need to decide what a transaction is in our system, before discussing the applicability of the atomicity, consistency and isolation properties. There are three candidates for a transaction:

1. a single message of a single feature
2. all messages of a single feature
3. a complete solution

All three lead to very different interpretations of the properties, and we consider them in turn.

1. A single message of a single feature

Assuming a transaction to be at the level of a single message, we do not obtain consistency: either all messages of a feature must be considered or none. Isolation is not possible, when messages of different features are interleaved, the order of these messages might matter whereas isolation would require that the order is irrelevant.

2. All messages of a single feature

Should a transaction be at this level, consistency is granted and isolation is also provided. However, we discard many possible solutions, as interleaving is now at a feature level (or coarse grained) as discussed earlier.

3. A complete solution

Having a complete solution as transaction satisfies the consistency requirement. However, isolation becomes meaningless, as no possible concurrent actions are available.

We can see that independent of the definition of a transaction one or more of the ACID properties is violated. Hence, one might question that we can use the term transactional to describe our approach. However, we believe that due to the close resemblance of the behaviour of the construction and error recovery this term is indeed justified.

	CFB	CND	CNDB	CT	CW	GR	RC	RBWF	SB	TL	TCS	TWC	VMR	VMP
CFB	-	U			T			T			U			
CND		-			T	U		T		U		U	U	U
CNDB			-	-		-		-						
CT				-			U		U	T	-	-	-	T
CW					-	-		T			-	-	-	-
GR						-		o		-	-	T	-	-
RC							-		T	o		U		T
RBWF								-			-	o	-	-
SB									-	o		U		T
TL										-	U	T		T
TCS											-	-	-	-
TWC												-	-	T
VMR													-	-
VMP														-

Tab. 7.1: Results from the Running Example

7.5 Analysis of Scenarios

7.5.1 Running Example: Multiple Point of Call Control

The number of potential combinations of features and the states they can be in is large. For the purpose of the analysis we have considered a small selection of scenarios. In particular, we consider two-way combinations of all features, but do so only in selected states and with selected input messages. In MPCC settings all considered features are located at the same user, features at the remote end can not be influenced. Thus, in this analysis the features are indeed all located with the same user. Interactions between features on different call sides cannot be detected using the given detection method as not enough information is available (we will discuss this later at the example of call forwarding loops). However, this is not a drawback of the resolution mechanism, it merely identifies a weakness of the detection method. 105 two-way cases have been analysed and the results are summarised in Tab. 7.1.

We distinguish five (distinct) results, marked with symbols as follows.

1. "-": an interaction has been detected, but all features request the same action,
2. "**T**": a technical interaction has been detected, removal of bad solutions is required,
3. "**U**": a user intention violation has been detected, we allow all features to proceed,
4. "**o**": an interaction has been detected with the features requesting different actions, but the resulting behaviour does not constitute a user intention violation or technical interaction,
5. " ": no interaction has been detected between these two features.

Clearly, when all features request the same action (case 1), we do not need to resolve an interaction. We can let all features continue, as achieved by choosing branches

satisfying the largest number of features. Similarly, when an user intention violation is detected (case 3), we have made the design decision to let all features proceed. This is achieved in the same way as for case 1. When no interaction has been detected (case 5) or an interaction has been detected but the behaviour is "desired" (case 4), the trace satisfying the largest number of features is again the appropriate solution.

The most interesting case is that of the detection of technical interactions (case 2). Here a resolution using the message dependent rule (i.e. pruning) is required. As described in section 4.2, offline analysis of the solution space is used to identify undesired behaviour. Clearly, some domain knowledge is required to know what is considered undesired. We considered, amongst others, contradicting billing messages and contradicting busy treatments to be undesired. The following pruning rules have been identified:

1. a trace containing *o_alert* and *store_read* in any order is undesired, we either want to initiate a call or query the voicemail feature,
2. a trace containing *billing_split* and *billing_reverse* in any order is undesired, the two billing messages contradict each other,
3. a trace containing *o_alert* and *announce(wrongpin)* in any order is undesired, the call should not take place if the PIN was incorrect,
4. a trace containing more than one of *o_inform(cwhold)*, *o_notify*, *o_inform(ringback)* or *o_inform(callminder)* is undesired, as they are all contradicting busy treatments.

The regular expression describing the above rules is rather complex as it needs to take into account all other messages. For simplicity let *non-oalert-stread* be the set of all messages that are not *o_alert* or *store_read*. Similarly we define the message sets *non-billsplit-billreverse*, *non-oalert-announce* and *non-busytreatment*.

The pruning expression can then be formulated as:

$$
\begin{aligned}
&(o\_alert.\textit{non-oalert-stread}^*.store\_read) \mid \\
&(store\_read.\textit{non-oalert-stread}^*.o\_alert) \mid \\
&(billing\_split.\textit{non-oalert-announce}^*.billing\_reverse) \mid \\
&(billing\_reverse.\textit{non-oalert-announce}^*.billing\_split) \mid \\
&(announce(wrongpin).\textit{non-oalert-announce}^*.store\_read) \mid \\
&(announce(wrongpin).\textit{non-oalert-announce}^*.o\_alert) \mid \\
&(o\_inform(cwhold) \mid o\_notify \mid o\_inform(ringback) \mid o\_inform(callminder)). \\
&\qquad \textit{non-busytreatment}^*. \\
&\qquad (o\_inform(cwhold) \mid o\_notify \mid o\_inform(ringback) \mid o\_inform(callminder))
\end{aligned}
$$

Having implemented these rules we were able to resolve all interactions in the two way scenarios, whereby in all cases the maximal number of features was allowed to proceed.

In general this was 2, apart from technical interactions, where one feature had to be blocked. In order to show that the approach does indeed work for n-way interactions, we have performed a number experiments with more than two features. Again, using the same rules as for the two-way cases, all interactions have been resolved in such a manner that as many features as possible could proceed. Some of the cases considered are as follows:

- 7 Fs BCS-CT-RC-SB-TL-TWC-VMP: Basic Call Software, Call Transfer, Reverse Charging, Split Billing, Teen Line, Three Way Calling and Voice Mail (Playback). The trigger event was *dial*, to which all features responded in their initial states. The constructed solution space had 44066 nodes, construction was completed in 32.24 seconds. The on-the-fly method provided a resolution in 18.96 seconds (of which 17.56 where consumed by *findfinal*).
- 6 Fs BCS-CFB-CND-CT-TL-VMR: Basic Call Software, Call Forwarding Busy, Calling Number Delivery, Call Transfer, Teen Line and Voice Mail (Recording). The trigger event was *i_alert*, to which again all features responded. The solution space had 866 nodes and was constructed in 0.52 seconds.
- 5 Fs BCS-CT-CW-GR-TWC: Basic Call Software, Call Transfer, Call Waiting, Group Ringing and Three Way Calling. The trigger event was *i_alert*, to which all features responded. The solution space had 6810 nodes and was constructed in 3.76 seconds.
- 4 Fs BCS-CT-CW-TWC: Basic Call Software, Call Transfer, Call Waiting and Three Way Calling. The trigger event was *flash*, to which all features responded. The solution space had 162 nodes and was constructed in 0.06 seconds.

Further details of the runtimes can be extracted from Fig. 7.3, which contains the results of the above cases. Note that again all features were located with the same user, and the features initial states have been chosen such that interesting behaviour occurs. The examples show that the solution space can become rather large, however the number of rules required to successfully resolve the detected interactions is very small. Furthermore the observed construction times are acceptable, especially if we take into account that pruning is efficient and the on-the-fly approach allows us to reduce the size of the constructed solution space significantly.

Including basic call as a feature proved to be acceptable. However, basic call often interacts with other features. We may not for example want the user to receive the response from basic call, but rather the response from the features. However, simply pruning traces in the same way as when interactions are detected is not desirable – often basic call provides the necessary trigger. Let us consider this situation in more detail.

Basic call may produce an *o_busy* message, which subsequently becomes redundant because a feature responds to this. In this case it would be desirable to remove

o_busy from the sequence of messages associated with the user. However, it cannot be removed from the messages used to reinstantiate the features as basic call should indeed be moved into the state it would have reached after sending the message (the respective features must also be moved to their new states).

It might be desirable to adapt the *commit* function slightly. Recall, once a resolution has been chosen, *commit* will reinstantiate all features in their desired states and issue the responses to the user. The solution is to adapt *commit* to scan the sequence to be committed for the occurrence of an *o_busy* message and if this is found, to identify whether any of *o_inform(cwhold)*, *o_notify*, *o_inform(ringback)* or *o_inform(callminder)* occur subsequently. If one or more of these messages are detected, *o_busy* should not be committed to the user, but rather should be omitted from the sequence. Note that this mechanism has not been implemented in the current *commit* function and that interactions of this form between basic call and other features have not been included in Tab. 7.1.

7.5.2 DESK Features: Single Point of Call Control

DESK uses a single point of call control model. In order to compare our results to those obtained by Marples [Mar00] achieved using DESK we must make some minor changes to our implementation.

A single point of call control (SPCC) means that features connected to the feature manager can belong to any user in the system. In contrast, in the multiple point of call control (MPCC) setting, which we used to achieve the results in section 7.5.1, all features connected to the feature manager are subscribed by one user (the remote end of the call is considered to be completely independent, as it potentially is hosted on a different switch over there is no control). The impact of this difference is best shown with an example: Assume that two users A and B subscribe to *call forwarding unconditional*. A attempts to ring B, thus producing an *incoming_ring* message. In the MPCC setting the feature manager at B's end of the call will issue this message to the subscribed features resulting in the expected forwarding initiated by B. However, in the SPCC setting both A and B's *call forwarding unconditional* will receive the trigger and both will respond. Clearly we do not want A's *call forwarding unconditional* to react to incoming calls to B before they have been forwarded.

To resolve this issue, our message format is extended by two fields: the source and destination of a message (thus including the same information as DESK's messages). Features perform a check on incoming messages as to whether the id of the subscribing user is the same as the destination of the message. In the above case, A's *call forwarding unconditional* will recognise that the recipient for the message is B and will not be triggered. Making this change in the model reflects the solution that has been implemented in DESK.

In most cases two features interact because they either share a trigger event or one feature triggers the other – i.e. they are either *shared trigger interactions* or *sequential*

action interactions. However, some features only interact in the presence of a basic call model when the user is busy. Marples deals with this by placing the respective user in a busy state (e.g. by going offhook). We deal with this by including a very simple basic call model in parallel with the features. This basic call model consists of a state machine with one state, upon receipt of an incoming ring the *term_busy* message is produced. Clearly this simulates the relevant part of DESK's full basic call model when a user is busy. All these cases result in sequential action interactions.

Consider the scenarios involving basic call individually:

CFB-CFB Assume users A and B subscribe to CFB; user A's CFB forwards a call to user B while user B is busy. B's basic call produces the *term_busy* trigger thus triggering B's CFB.

HL-CFB Assume user A subscribes to HL to B, and B subscribes to CFB. If B is busy B's basic call produces the *term_busy* trigger upon detection of A's HL call attempt.

RCI-CFB B's RCI returns a call to A who subscribes CFB. If A is busy, the RCI call triggers A's basic call to produce *term_busy*, thus triggering CFB.

HL-CW is similar to HL-CFB (assume B subscribing CW instead of CFB).

TCS-CW Assume A subscribes to both features. TCS is triggered upon an incoming ring message, the basic call produces *term_busy* upon an incoming ring on a busy line. Feedback of the *term_busy* triggers CW.

Using the adapted model, we have considered all features pairwise obtaining comparable results. Table 7.2 contrasts our results to those obtained by Marples [Mar00]. Each entry is composed of a left and a right symbol, either or both might be blank. A blank simply means that no interaction has been detected. In addition to blank, the left symbol can be + or -, reflecting whether Marples detected an interaction or Marples did not detect an interaction, but expected one. The right symbol is either blank or x, where x means that we detect an interaction.

Unsurprisingly, the detection results are nearly identical. The reason as to why they are not completely identical is undecided as certain DESK features showed erratic behaviour. The version of DESK that we used produced results slightly different to those reported by Marples. For example, we were able to detect an interaction between RCI and TCS, as one would expect. Assume that B subscribes to RCI and A subscribes to TCS barring calls from B. When A calls B while B is busy, B's ringback will initiate a call that will then be barred.

We were able to resolve all interactions successfully, employing message dependant pruning. Occasionally, removing duplicates and extracting traces with most features satisfied left a choice between two (or more) traces; we simply extract the leftmost trace. Thus all interactions were resolved successfully, again allowing both features

	CFU	DND	OCS	TL	CFB	HL	TCS	CW	RCI
CFU	+ x	+ x	+ x		+ x	+ x	+ x	+ x	+ x
DND					+ x	+ x	+ x		+ x
OCS					+ x	− x			− x
TL						+ x			− x
CFB					+ x	+ x	+ x	+ x	+ x
HL							+ x	+ x	+ x
TCS								− x	x
CW									+ x
RCI									

Tab. 7.2: Detection Results of Marples and Reiff-Marganiec

to proceed when the interaction was considered to be a User Intention Violation (e.g. CFU-DND or HL-CFB).

The message dependent rules require a regular expression describing bad patterns. We can use domain knowledge to identify a number of behaviours that we view as undesirable, at least between two stable states:

1. connecting a user to two different resources,
2. routing a call to two different locations,
3. routing a call away from A and still changing A's local state,
4. routing a call away from A and connecting A to a resource,
5. changing A's state and connecting A to a resource.

A more technical analysis of the above cases reveals that they involve one or more of the following messages:

- *send_to_resource, A, res* results in user A being connected to resource *res*, e.g. an announcement or a busytone
- *routing, A, B* results in a call being routed from A to B
- *move_to_state, A* results in A's call software being moved into a new state, used by call waiting

We describe two of the above cases as regular expressions, using the messages as defined.

Example 7.5.1 (Connecting a user to two different resources) Let *str_xxx*11 be the regular expressions for the *send_to_resource* message that connects user 1 to a resource. As example *str_nodis*11 = *MkRE* (*Snd*, *Msg_send_to_resource*, *Nodisturb*, 0, (1, 1)).

$$
\begin{aligned}
tworesources = &(str\_nodis11.str\_ocs11)|(str\_ocs11.str\_pin11)) \\
&(str\_pin11.str\_tcs11)|(str\_tcs11.str\_nodis11))) \\
&(str\_ocs11.str\_nodis11)|(str\_pin11.str\_ocs11)) \\
&(str\_tcs11.str\_pin11)|(str\_nodis11.str\_tcs11)))) \\
&(str\_ocs11.str\_tcs11)|(str\_pin11.str\_nodis11)) \\
&(str\_tcs11.str\_ocs11)|(str\_nodis11.str\_pin11)))
\end{aligned}
$$

Example 7.5.2 (Routing a call away from A and connecting A to a resource) In this example $str11$ is the regular expression matching the single occurrence of any send_to_resource message from user 1, $routng1x$ is the regular expression for the single occurrence of a routing message routing calls from 1 to x. Also $nonrtngstr11$ is the regular expression describing the occurrence of none or many of the remaining messages.

$$
\begin{aligned}
rtngandstr = &(str11.nonrtngstr11.(routng11|routng12|routng13))| \\
&((routng11|routng12|routng13).nonrtngstr11.str11)
\end{aligned}
$$

Note that the regular expressions include all possible instantiations of the parameters of a message that occur in our examples. In this case an extension to the regular expressions could be considered, where the matching does not compare the whole message but rather the relevant parts.

Using only these five simple regular expressions and the message independent rules we were able to resolve all detected technical interactions automatically. The remaining interactions have been resolved by using only the message independent rules as described earlier.

7.6 Appropriateness and Suitability

The evaluation performed so far considered whether the specification was met by the implementation. We have also considered scalability and shown two case studies that provided empirical results. Now we step back from the technical details to consider whether the approach is suitable and appropriate to detect and resolve interactions.

In the case studies, we have been able to detect and resolve interactions in single and multiple point of call control settings automatically. In particular, the best possible resolutions (based on our understanding of the features) have been found for all detected interactions. The patterns describing bad behaviour as used by the pruning algorithm have been relatively obvious and only very few such patterns were required. Our understanding of the semantics of the existing messages made the formulation of the rules possible.

As features are distinct in such a way that they do not allow general reasoning, e.g. based on their structure, evaluating the approach using case studies is the most appropriate way of convincing oneself of the suitability.

As user intentions are often ambiguous, we have concentrated on technical interactions. Recall that a technical interaction is defined to occur when several features triggered by the same response or features triggered by an earlier response, request for the call to be continued in distinct, non-unifiable ways (Definition 4.4.1). In the definition this was further explained as the absence of a system state which satisfies the behaviour of all requests (similar to type I interactions as described by Hall [Hal98]).

However, in the detection method developed by Marples and refined in this thesis, and the presented resolution techniques, we assume that no knowledge of the state of the system is available. So, how can we relate our approach to the definition of technical interactions? Consider the two components of the approach, the construction and the resolution.

Solutions. The construction method, i.e. identifying the solution space, produces all possible behaviours. Multiple (distinct) solutions exist precisely because several features have responded to a trigger event or because features have responded to a fed back response. However, this alone does not allow for the detected behaviour to be classified as technical interaction: It is possible that the requested call continuation behaviour is unifiable, potentially leading to an user intention violation, or, more interestingly, leading to behaviour that is completely acceptable (such as delivering a voice announcement and also displaying the same information). The potential of the latter is the core motivation for exploring all possible interleavings.

In SPCC settings more information is available and the control of the feature manager extends to the whole call, rather than just a call leg (as in MPCC). This allows one to detect more interactions than in the MPCC setting. In particular, the MPCC setting does not allow one to detect call forwarding loops using our mechanism. In order to detect forwarding loops some form of call history, e.g. additional information attached to the messages, would be required. A forwarding message would carry as an additional argument the history of all previous forwards, which then can be analysed w.r.t. whether the call has already been forwarded from this location.

Resolutions. The aim of the resolution method, in particular the pruning rule, is to ensure that technical interactions cannot occur. Namely, applied to the solution space, all those solutions that lead to requests for continuing the call in non-unifiable ways are removed. Thus, while our detection method does not distinguish the different classes of interactions, the resolution method deals with the class we are interested in, i.e. the technical interactions.

The quality of the resolutions depends on the knowledge of the message semantics. However, even a general understanding of the messages is sufficient. For any system under considerations, it must be assumed that this general understanding exists, as otherwise enhancement is questionable even in the absence of the feature interaction problem. In the context of telecommunications systems, the general understanding of a message is usually obvious, e.g. *dialtone* means that the user now can dial a number. In general, messages that lead to actions that must be consistent for a user should be used in a consistent fashion, i.e. their semantics is fixed.

A rule set is considered complete if all interactions can be resolved. However, there is no generic method of identifying a complete rule set (due to the diverse nature of features). This is a drawback, as an incomplete rule set can lead to a trace being identified as resolution when the trace is not a resolution. However, this is only a minor drawback, as understanding of the messages and the in the studies encountered message sets are relatively small it is possible to consider sequences of messages and deciding whether they are acceptable.

For larger message sets, this manual approach might prove impossible. However, all that must be automated is the analysis of sequences formed from messages from the message set. It is not required to know details about the features. Thus formulating the rules is independent of the features.

In addition to the rules resolving technical interactions, we can also define rules that exclude message sequences that can be considered to cause user intention violations. However, these will be very crude in that they might not match every users expectations – they merely reflect the view the rule designer has on the action of the involved features and whether their interaction is considered undesirable.

Complexity. Overall complexity did not initially look promising. However, considering the different parts of the approach that introduce high runtimes, we find that the most expensive part of the process is the generation of the deterministic finite state automaton (DFA) used to match the pruning pattern. Fortunately, this DFA is only constructed at startup of the system and when new rules are added (which should only happen very infrequently). Further, a significant improvement can be made on the matching algorithm using a hash table for the transition table of the DFA.

Solution spaces involving many features can grow large (we have seen an example with more than 40000 states earlier). However the construction is performed reasonably quickly and the pruning and extraction are not too time consuming as shown earlier. The on-the-fly approach combining pruning and construction rarely performs worse than the construct-prune approach (it does so when a feature does not contribute to a solution and thus duplicate traces are introduced) and provides a significant improvement if traces are long and can be pruned early in their construction. Longer traces are normally a result of many features interacting, thus the on-the-fly approach becomes more viable as the number of features increases. Note that it is crucial to use a hash table for the transition table of the DFA used by pruning to increase performance.

Scalability. Details of scalability have been discussed previously. Evidence so far suggests that the approach is scalable, however the suitability of this approach for use within an operational telecommunications switch remains the subject of further work.

Summary. The rule based resolution approach is desirable as it addresses the problems posed in the motivation for this work. In particular it provides a solution to the feature interaction problem in the context of legacy and third party features which no other currently available technique offers. As discussed, detection and resolution are

performed without knowledge of the internal behaviour of the features and resolution requires only an understanding of the semantics of the exchanged messages.

7.7 A Hybrid Approach – Revisited

In Chapter 4 we introduced the idea of a hybrid approach for feature interaction detection and resolution.

The main ideas discussed involved the development of an initial model, applying offline analysis to the behaviour and identifying a set of resolution rules. This rather formal part was then seen to be integrated within the DESK testbed. In an iterative process we learn from experiments in DESK, i.e. weak points of the resolution approach, and would then refine the model and resolution rules.

We have developed a detailed specification and model implementation of the underlying system. The model is the result of many failed attempts, mostly concerned with finding the right notation for modelling. However, those "dead-ends" have led to a sound understanding of the relevant issues:

- We can only assume knowledge of observable behaviour of the features – the impact of this assumption influences the detection and resolution mechanism fundamentally and also impacts on correctness proofs.
- The model must be relatively close to the real system – especially w.r.t. the feedback mechanism, but abstract enough to hide the implementation details.
- The call model, SPCC or MPCC, plays a major role as to what can and cannot be done, and influences the architecture of the model.

The specification of the solution space (Chapter 5) provides a clear understanding of possible solutions and hence resolutions. We can have two distinct classes of methods for reducing the solution space: pruning and extraction. The former being dependent on the semantics of the messages the latter being more general. Our model was built using Haskell.

We will now briefly explain how an integration with DESK would be performed. DESK's feature manager needs to be extended at different stages. At startup of the system the DFA corresponding to the provided regular expression must be computed. The further changes depend on the chosen approach: construct-prune vs. on-the-fly.

To integrate the construct-prune approach, DESK's current construction mechanism can be maintained. The change is, that rather than presenting the constructed solution space to the operator, it is passed to the automatic resolution method. Once a resolution has been chosen, the system proceeds as previously.

The on-the-fly approach is more intrusive, as a change to the method of construction is required. In order for this approach to be integrated the current construction and resolution mechanism of the DESK feature manager would need to be replaced. The new construction algorithm would be exactly as presented in the model. Again the resolution would be passed back to the system at the same point than it was after the operator had chosen it.

We had proposed to integrate our resolution method into the DESK testbed in order to explore the success. We decided that this is not desirable at this stage for several reasons: the testbed itself proved to be fairly fragile, and furthermore it was restricted to a single point of call control setting. On the other hand, the close proximity of the model and the testbed, and the fact that modelling in Haskell allows us to generate executable prototypes, suggested performing experiments on the model.

Overall, we conclude that the initial idea of a hybrid approach proved fruitful, but the details of the approach that were described earlier required certain adjustments. Mainly, we did not improve the resolution rules by new knowledge gained from observing the system with an initial resolution strategy and we did not integrate the develop resolution technique into the DESK testbed.

7.8 Summary

We have discussed the correctness of the approach with respect to the specification and have considered the complexity. An analysis of scenarios placed the theoretical considerations in the context of realistic examples taken from the running example and the DESK testbed. The reliance on a semantics of the messages as well as the subjective decision as to what constitutes bad behaviour might be seen as weaknesses of the resolution method. We conclude that the approach is indeed suitable to fulfil the set aims. Finally, we reflected upon the hybrid approach and the integration of the developed approach in the DESK testbed.

Chapter 8

Conclusions and Implications

8.1 Introduction

Our aim was to show the desirability and feasibility of an approach to detect and resolve feature interactions in evolving telecommunications systems. We have outlined a number of smaller aims and formulated some objectives (in section 1.2) describing how we intended to achieve the principal aim.

We now reflect on the presented work, considering if and how the aims were met. We then consider implications of this work for both the theoretical and practical aspects of the research area. We also discuss transferability of the approach to other areas, as well as its limitations.

Ideas for further work will be presented. In particular, we consider the shift in technology that is presently taking place and discuss its impact. We reason that the technology shift has and will have a significant impact on telecommunications systems. However, this shift neither removes the feature interaction problem nor invalidates our approach.

8.2 Reflection on Research Problems

The overall aim of detecting and resolving feature interactions was split into several subgoals. The subgoals are part of three groups of research questions: the first concerned with possible solutions, the second concerned with good solutions (or resolutions) and the third applicability and suitability of the approach.

Solutions. We have defined a solution to be a trace of one or more features running concurrently. The number of potential solutions can be very large, it is bounded by the number of features and the length of their responses. All these solutions are found by a feedback process following Marples's idea [Mar00]. However, we identify even more solutions than Marples, in that we allow interleaving on a message basis rather than on a feature basis – we refer to these as fine and coarse grained interleaving respectively. When considering resolutions we need to discuss whether it is meaningful to have these distinct kinds of interleaving.

The feedback process is a runtime detection method and does not require any semantic information about the messages, it is simply based on responses of features to trigger

events. We considered that two (or more) features might be triggered by the same event or that a feature might be triggered by a fed back response from another feature. Any combination of these two triggering possibilities is also considered.

Building on the definition for solution, we identified the solution space – i.e. the set of all possible solutions and also provided a way of constructing the same at runtime. A quantification of possible solutions has been provided in this context, but note that there is no quantitative relation between the number of features and the number of interactions. This is due to features reacting different depending on certain events and often behave different in the presence of other features. In addition, the internal behaviour is not known, so we do not know in advance how a feature reacts in a given situation. The analysis allowed to place an upper bound on the number of solutions. This upper bound is provided by the number of features and the number of messages they send as response to a trigger, though the combined length of responses is limited by the exploration algorithm making this bound firmer.

Resolutions. Understanding solutions was the basis required to develop the resolution approach. The concept of resolutions was made precise as "good" solutions.

Our approach distinguishes two broad categories of resolutions rules: message dependent and message independent. The rules are used to define pruning and extraction operations. Extraction simply maintains solutions that we deem better than others, based on notions such as satisfiability of features or priorities. For this to work, it is only necessary to know which messages stem from the same feature.

Pruning rules requires semantic knowledge of the messages in order to specify patterns describing bad behaviour. Regular expressions are the mechanism used to describe rules expressing bad behaviour. We discussed that regular expressions are sufficient to describe the behaviours that we wish to exclude. Pruning removes solutions that contain patterns described by the regular expressions.

By identifying several rules based on the case studies we contributed the basis of a description of undesirable behaviour as it occurs as part of possible solutions. This allows to eliminate undesired behaviour at runtime in a more general setting than has been possible so far. Previous work makes use of solutions for known conflicts or human input to resolve detected interactions.

The message independent rules alone are capable of resolving some interactions. However, the pruning rules add significant strength to the resolution mechanism in that they allow for very specific behaviour to be excluded. It is here that the fine grained interleaving gains relevance: it is hoped that sometimes it might be possible to have two features active together when their individual responses are interleaved, whereas they would not be able to cooperate if the interleaving is on a feature basis.

The major drawback of the pruning rules is their formulation requires domain knowledge and that we do not have a general way of showing that the rule set is complete. The rule set is dynamic, that is the rules are adapted when required. This was the motivation for the iterative process described in Section 4.2. In the absence

of domain knowledge a reliance on the semantics of the messages is the only way to formulate meaningful pruning rules. However, the set of rules will almost certainly be weaker than one found assuming domain knowledge and is much more dependent on a better understanding of the message semantics.

Applicability. We have integrated pruning with the construction of the solution space, we referred to the result as the on-the-fly approach. Trials showed that the on-the-fly approach provides scalability. Considering the construction times in conjunction with the runtimes for pruning and the on-the-fly method, the resolution overhead is quite small (or non-existent). We note that the runtime of our trials are not too promising, but this is largely due to the usage of Haskell[1]. Furthermore, more efficient data structures will improve performance as discussed earlier. As Marples found the runtimes for his construction method acceptable, we can conclude that, since the construction method is essentially identical and the resolution overhead is minimal, our overall runtimes must be so as well.

Our evaluation was performed on sets of up to 12 features simultaneously present in the system, but we cannot limit the maximum number of features or even guarantee that for every feature set a minimum can be assured. This is due to the diverse nature of features and the absence of a generic description of features. As discussed earlier both the number of features and the number of messages in the system are important factors to be considered when discussing scalability.

The number of detected interactions depends largely on the underlying system architecture. When single point of call control is assumed, more interactions can be detected. However, even in multiple point of call control systems we were able to detect interactions between features subscribed by one user.

The resolution technique works in both settings but as it crucially depends on the earlier detection of interactions it can only be as good as the detection method.

In order to achieve better performance in multiple point of call control settings, more information exchange between the control points is required. New telecommunications architecture (as discussed later) allow for this additional exchange, thus strengthening the work done here.

We have shown that the developments in the telecommunications area towards open markets with multiple vendors as well as legacy equipment require runtime solutions for the feature interaction problem. Our method, which is based on a transactional approach, provides solutions as it allows to detect and resolve feature interactions in the legacy context. The evaluation and discussion concluded that it is indeed feasible and desirable to have such an approach.

[1] [Bag01] provides a benchmark for different programming languages. On average, programs compiled using `ghc` are 6 times slower than their `gcc` counterparts

8.3 Transferability to Other Areas

Blair and Blair [BB01] presented a method for dynamic Quality of Service Management using a system integrating controllers and monitors. In their system, the monitors and controllers provide additional functionality to a basic service, and can as such be regarded as features. Furthermore, communication between these components is via message passing. In a system with multiple monitors and controllers, feature interaction can occur when controllers compete for bandwidth. Our approach has been applied to detect and resolve interactions in the area of Quality of Service Management. The results are reported in [BR01].

The Advanced Separation of Concerns Workshop (part of ECOOP 2001) included a discussion group concerned with Feature Interaction for ASoC models. This highlights the fact that the feature interaction problem has been recognised in other areas. It remains to be seen whether the presented approach can be adapted to this area, as well as other component based systems.

However, we believe that the method can be employed successfully to detect and resolve interactions in component based systems, provided that the inter component communication can be intercepted, delayed and blocked and that sufficient information is conveyed in the messages. This claim is supported by Blair et al. [BBPE01]. They propose that a runtime approach to discover problems after reconfiguration and those unforseen at design time is necessary to resolve interaction in component based middleware.

In summary, the approach may be applicable outwith the telecommunications domain and presents a contribution to the feature interaction problem in component based systems.

8.4 Limitations

In a fast advancing area it is unlikely that one approach will be able to provide a general solution. We have outlined the setting for which our approach has been developed, namely emerging legacy systems and third party components in the telecommunications domain. We have shown that we have successfully detected and resolved feature interactions in our evaluation.

Our approach does have limitations, which have been highlighted earlier. They are mostly concerned with the architecture of the system, especially the communications mechanism and the ability to integrate new components. A system where the communications between the components can be intercepted, blocked and delayed together with the possibility to insert the feature manager component into the system is required. Not all systems will allow for this: the communication path might guarded in some fashion, e.g. encryption and time stamping, such that any delay or blocking

of messages is seen to be intrusive and the communication will be stopped as being insecure. The approach will not be applicable to such systems.

The feature manager must be able to temporarily block message exchanges and explore feature behaviour without the results being committed to the system. This might not be possible, and more significantly the time required might not be available in every system. The latter can be caused by time critical transmissions, where a response is required to happen within a certain time scale that prohibits anything but an immediate reply. If features have side effects, such as changing global data and the changes cannot be rolled back when the features are rolled back, the exploration mechanism is unusable as partial information is committed leading to inconsistent information in the system. Clearly the approach will not be applicable in this situation.

We have assumed that features do not have any side effects, which is necessary for the rollback mechanism between stable states. Once we have committed to a resolution we reach a new stable state. There is no reason why a feature shall not be able to continue to progress from a new internal state once we have reached this new stable state (and in fact several of the example features do this, e.g. Teen Line). However, rollback only works between stable states, and there is no mechanism to reset features once they have been committed to. From a user point of view this is not a limitation, in fact allowing rollback across stable states would be confusing for a user.

To ensure consistency in the case when features go across several stable states we can see two solutions: a history or look-ahead solution. The former requires for a history of which features have been committed to be kept and if the same feature reacts again in a consecutive resolution attempt it shall be given a guarantee to continue. This opens the question as to when we can consider the behaviour of a feature to have terminated, i.e. how many steps into the past do we need to consider. The latter would require the feature manager to be able to predict user inputs and explore the future behaviour with respect to further inputs. However, we cannot guarantee the prediction to be correct and thus might fail to resolve the problem. Furthermore, the complexity of the resolution process is increased. In the current solution neither of the approaches to deal with features crossing stable states was required and hence this has not been considered in detail.

The success of our approach depends on the understanding of the exchanged messages. As discussed in Chapter 7, this is not unreasonable. Most communicating systems operate to some protocol which provides the semantics of the messages and telecommunications systems form no exception. If the understanding does not (or only partially) exist the approach might still be applicable. However, in this case a reduction in the quality of the resolutions must be expected.

8.5 Further Research

We have presented a method to detect and resolve technical feature interactions in evolving telecommunications systems. We concentrated on technical interactions.

Several aspects for further work can be identified. We consider possible improvements to the approach itself, the evaluation of the approach in an operational system, a transfer to other areas and the relation to new technology. This section explores these issues in more detail.

To improve the runtime performance even more than with the on-the-fly method, a heuristic is considered. The idea is: certain message sequences might always suggest a failure in the future without themselves being bad. Assume a sequence s_1 which is not bad, and a sequence s_2 which is bad. A heuristic might say that s_2 always occurs some time after s_1, thus the pruning can be performed when encountering s_1, rather than awaiting the offending sequence s_2. This allows for earlier pruning and hence for better performance. Clearly such heuristics require a very good understanding of the underlying messages, especially which messages are only meaningful if they occur in conjunction with each other. We can conclude, that to allow for heuristics the semantics of the messages is crucial.

When n features interact in an undesired way, then every trace containing these n features in addition to others will also lead to undesired behaviour. We could implement a mechanism to use this to our advantage. During the on-the-fly method, a log is kept as to when all traces involving certain features have been pruned. It seems that, if a new trace is to be constructed involving all those features we can simply skip the construction, knowing that all traces will be pruned as well. This requires further investigation.

Pruning uses rules that describe certain patterns of messages to be undesired, and in our evaluation this was sufficient. However, one could imagine scenarios where new features introduce behaviour that contradicts the provided rules. An example would be that in general we do not desire to route a call to several locations simultaneously, which would contradict the basic behaviour of a group ringing or conferencing feature. The matching of the rules could be extended in such a way that when a pattern is matched by a sequence of messages stemming from one feature, this will be acceptable and only those where a number of features lead to the pattern are undesired. If a feature is composed of several sub-features then this approach is not feasible. However, it remains unclear whether combining distinct features to obtain a new service is desirable. This is because we cannot guarantee that all those sub-features are enabled and each interworks correctly with the remaining features in the system. It appears that the combination of features to obtain such a combined service should occur at a different level.

Our approach has been implemented in Haskell, which proved a good choice for the modelling and simulation, but clearly lacks in realistic performance. Thus, it would be desirable to re-implement the ideas using C to make use of the better performance attainable. We have chosen Haskell for reasons discussed earlier. Using a C implementation to show the functional issues discussed in this dissertation seemed to detailed and thus diverting from the main aspects. Also this reimplementation should consider more efficient data types, especially for the transition table of the DFA.

Further, incorporation in an operational switch is desirable, to observe the runtime behaviour in a natural setting.

In summary, we have identified the following areas for further study:

- heuristics for earlier pruning,
- automatic learning techniques to improve pruning rules,
- "undesired behaviour" produced by a single feature,
- features crossing stable states (c.f. section 8.4),
- implementation and evaluation in an operational system.

The first four are incremental improvements ideas to deal with situations that might arise in the future, the last would provide a better measure of feasibility.

Our approach deals with the present, i.e. legacy systems, but also addresses the deregulation of the telecommunications market with some of its impacts. However, the telecommunications area is currently advancing at incredible speed and many new issues are emerging. We now consider these issues in some more detail and reflect on the impact of the presented approach.

8.5.1 Technology Shift

We observe a general merging of what used to be separate services: the PSTN (primarily a circuit switched voice network), the internet (primarily a packet switched data network) and mobile networks. In the new combined communications network, features considering billing or call routing remain, and new features for areas such as quality of service management arise.

We identify two dimensions in the technological advance: new network architectures and protocols and the addition of a service layer on top of core networks. Both dimensions allow for ever more complex services by assisting service creation and providing larger capabilities than traditional telecommunications networks. Features, in this new context often referred to as service logic, become more distributed. We are approaching an area that can be described as "anything over IP", i.e. the core networks will be packet and not circuit switched, and there will only be one core network technology reducing investment for operators significantly. Service development is facilitated by richer protocols, such as SIP [HSSR99] or Bluetooth [SIG01].

Today's telecommunications market can be described as vertically layered, that is each technology – GSM (Global System for Mobile Communications), IN (Intelligent Network), ISDN (Integrated Services Digital Network) – has its own layering and own services. However, recent developments, especially the integration of the different network types mark a move towards horizontal layering (similar to ISO-OSI network

layering), where we distinguish a resource, service and application layer. The resource layer simply constitutes the physical network access, the service layer provides capabilities such as call control, security or mobility. Applications simply provide the user with a tool that makes use of several capabilities, e.g. a basic call service or a video conference call. Applications gain access to the service capabilities via predefined APIs, examples are Parlay [Par] or JAIN [JAI].

Kimbler [Kim00] predicts 4 areas of service interactions: between the core network technologies, between the core networks and the service layer, within the service layer and between the applications. Notably he does not mention interaction between the service layer and the applications, presumably as this should be avoided by the interface. Thus interaction detection and resolution stays prevalent in the emerging technologies.

As our approach facilitates the resolution of interactions at runtime, it can be included in the new networks and resolve interactions at the appropriate layers simply by observing behaviour at runtime and applying the presented detection and resolution mechanism. Clearly, scalability would need to be reconsidered in the context of probably much larger message sets. In addition, the new richer protocols allow for communication between feature managers at different ends of the call and thus facilitate the acquisition of additional information about the remote end. This allows to detect more interactions. Further, remote feature managers could be presented with resolution choices or even just be informed about locally performed resolutions, thus increasing the overall quality of resolutions. Note that, in this context, negotiation approaches gain a new importance.

As further work, it would be desirable to explore the capabilities of our feature manager approach in the new setting and also consider a combination of the negotiating agents approaches with the feature manager. This will be helpful when considering user intention violations, which are currently not dealt with. In order to resolve user intention violations it is required that individual users can express their preferences, which in current systems can only be expressed by prioritising features. Emerging systems allow for more powerful mechanism, for example call control policies.

In summary, it can be concluded that the new developments do not reduce or remove the feature interaction problem. Thus solutions for the problem will be required and our approach looks very promising because it is capable of working with minimal information (i.e. the semantics of messages and domain knowledge as to which message combinations result in undesired behaviour). Recall that no information about the internal behaviour of the features is required. We rely on the semantics of messages, thus it would be desirable to have agreed interpretations as to what a particular message means. The approach might prove even more useful in the emerging systems, where richer protocols allow for more information to be available via messages and we concluded earlier that more information leads to better resolutions.

Appendix A

Formal Description of the Features

A.1 Introduction

The description of the features uses non-deterministic state machines. In an operational system this non-determinism is removed by the environment: For example features might react to times of the day, feature internal timers or data within the features initialisation. As we are primarily interested in the observable behaviour, i.e. input and output, the features environment is not modelled leading to said non-deterministic finite state machines.

When modelling the feedback process features cannot be non-deterministic, as this would allow a feature to react to the same trigger with different responses each time it is presented with the same trigger in a feedback process. In order to resolve this, it was chosen to select the interesting behaviour of a feature when non-determinism occurs and model only that. Respective notes are made with the features concerned.

State labels in the following state machines a purely to ease understanding and to provide a reference, they are not used in the actual detection and resolution mechanism. Start states are denoted by a bold circle, in addition the state number is 0. Transitions are often labelled with multiple messages, the used notation will be summarised briefly. Labels are of the following form: "*input* / output". Inputs are printed in blue and italics, outputs black and upright. Some transitions do not depend on an input, they are controlled by the feature internally. In this case the transition label is simple "output". Transitions leading to a new state but not resulting in any output being produced are labelled "*input* / -". Finally, there might be more than one output message resulting from a single input, in this case ";" is used to concatenate the individual messages.

A.2 Basic Call

After being alerted a terminating user (see state BC 6) has three choices guarded by an input. Further the system might timeout, modelled by a transition that does not require any input. Note that this makes the feature statemachine non-deterministic, and hence the transaction *o_timeout.stopalert* has not been included in the Haskell model.

Fig. A.1: Basic Call Model

A.3 Call Forwarding on Busy

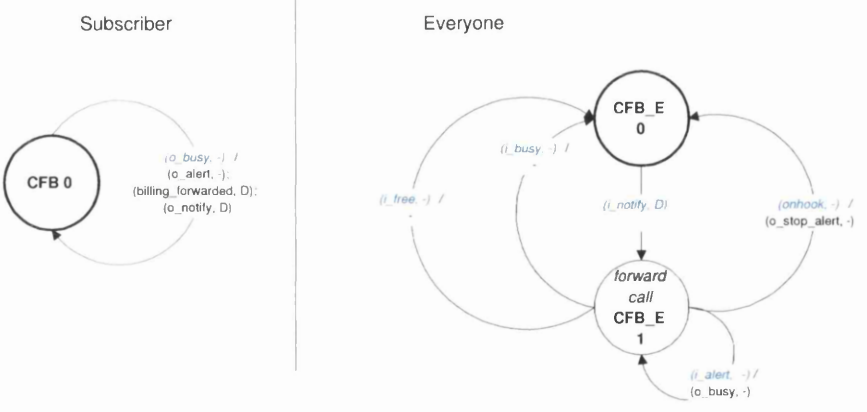

Fig. A.2: Call Forwarding on Busy Model

A.4 Calling Number Display

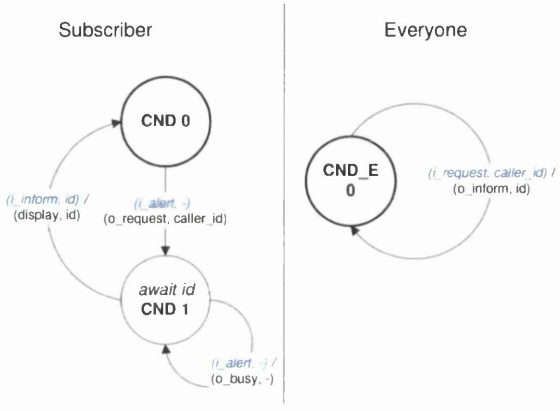

Fig. A.3: Calling Number Display Model

A.5 Calling Number Delivery Blocking

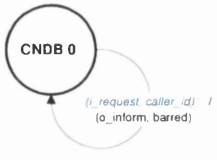

Fig. A.4: Calling Number Delivery Blocking Model

A.6 Call Transfer

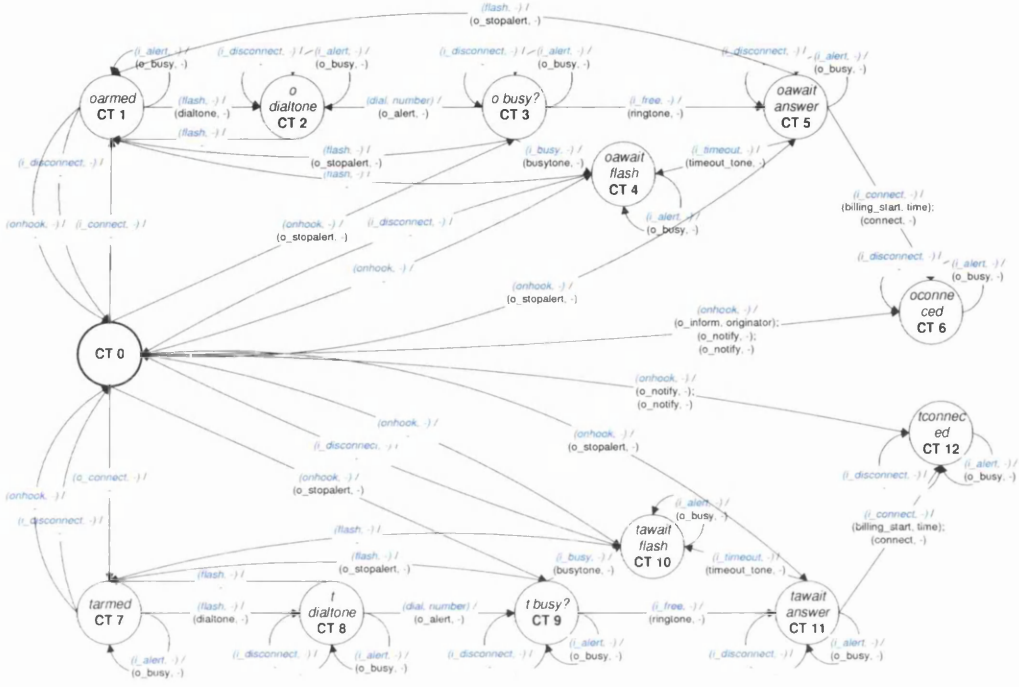

Fig. A.5: Call Transfer Model

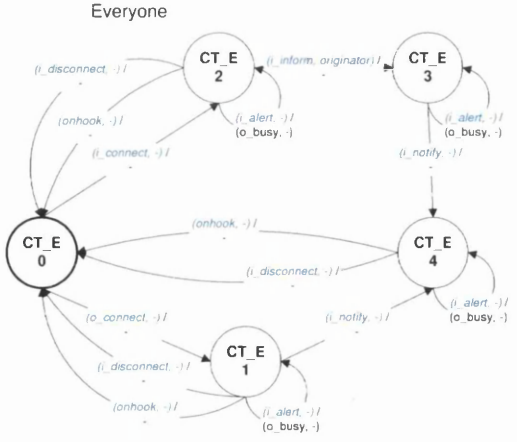

Fig. A.6: Call Transfer Model – Everyone

A.7 Call Waiting

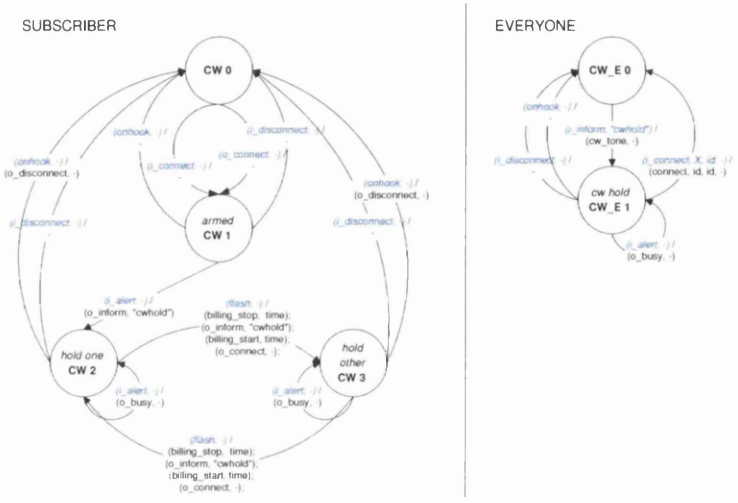

Fig. A.7: Call Waiting Model

A.8 Group Ringing

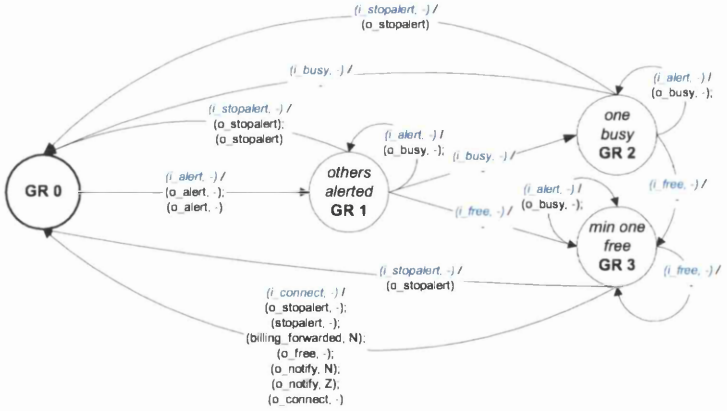

Fig. A.8: Group Ringing Model

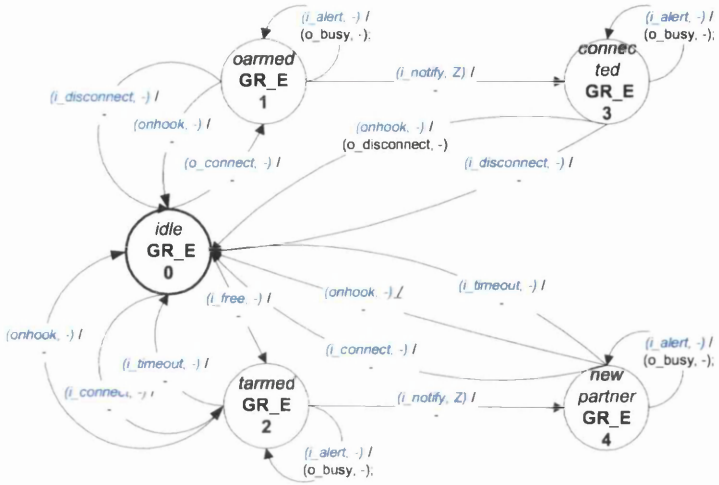

Fig. A.9: Group Ringing Model – Everyone

A.9 Reverse Charging

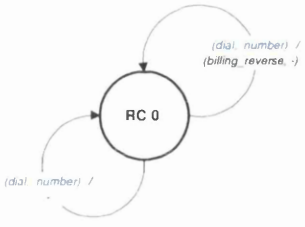

Fig. A.10: Reverse Charging Model

A.10 Ringback when Free

From the armed state (RB 1) this model has a number of non-deterministic choices. In reality these are deterministic, as they represent a decision on whether the ringback list is empty or not. In the Haskell model the three transitions from RB 1 to RB 0 are not implemented.

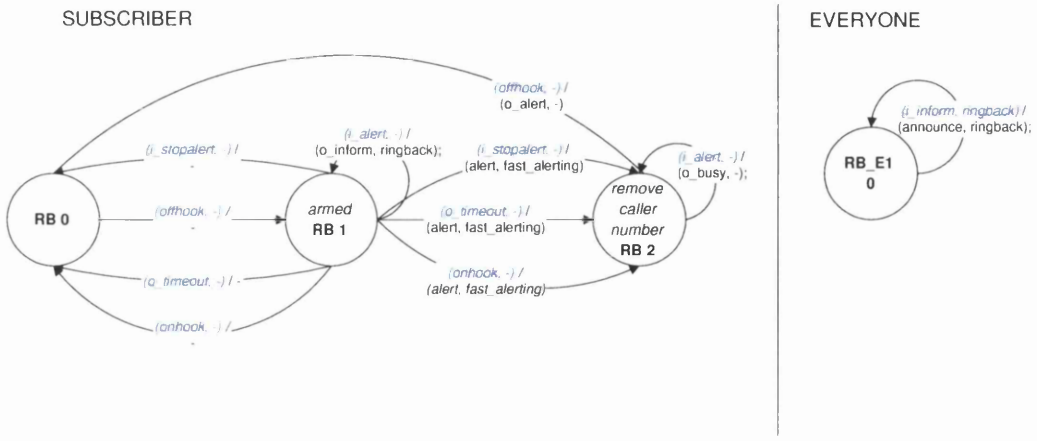

Fig. A.11: Ringback when Free Model

A.11 Split Billing

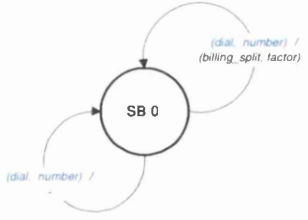

Fig. A.12: Split Billing Model

A.12 Teenline

This feature makes two internal decisions, hence the model introduces non-determinism. From state TL 0 two transitions triggered by *offhook* exist, representing the exclusive choices whether it is teen-time or not (the latter is represented by the transition from TL 0 to TL 0). Further from state TL 1 there are two transitions triggered by *dial*, the one from TL 1 to TL 0 is reflecting the entry of a correct PIN. In the Haskell model the *offhook* transition from TL 0 to TL 0 and the *dial* transition from TL 1 to TL 0 are not implemented.

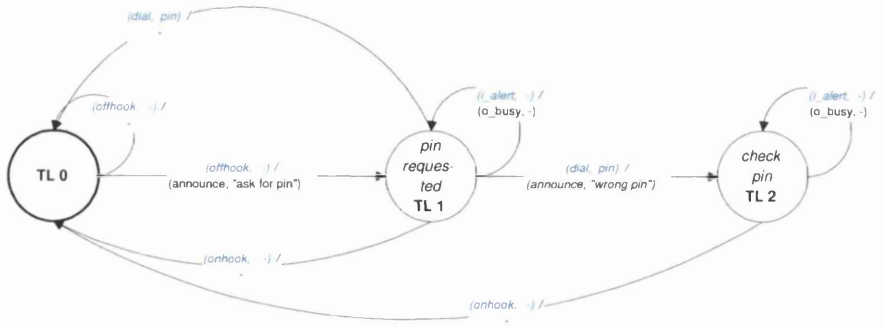

Fig. A.13: Teenline Model

A.13 Terminating Call Screening

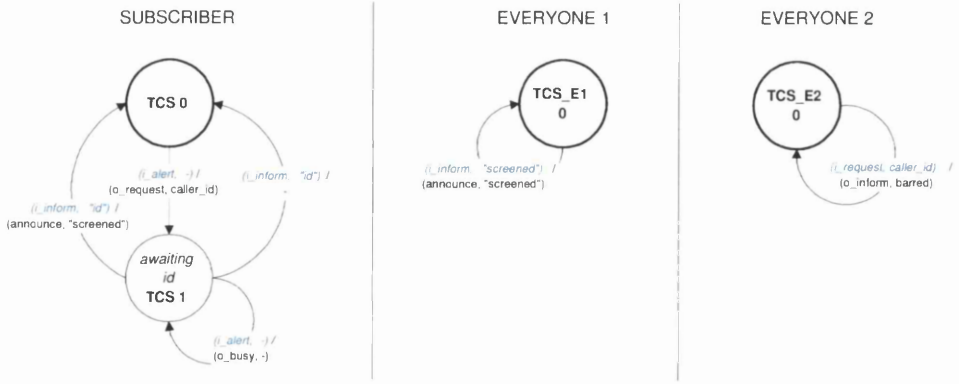

Fig. A.14: Terminating Call Screening Model

A.14 Three Way Calling

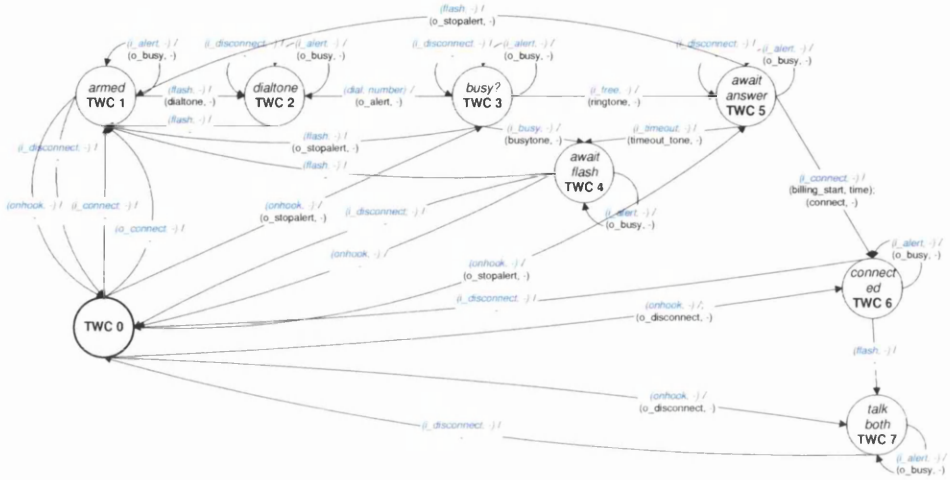

Fig. A.15: Three Way Calling Model

A.15 Voice Mail

The playback part contains three transitions triggered by *dial* from state VM_PB 0 representing the feature internal choice whether the dialled number is the activation code for playback and if so whether messages are available or not. The Haskell model does not contain the transition for "no messages available", neither the transition when the dialled number is not the activation code.

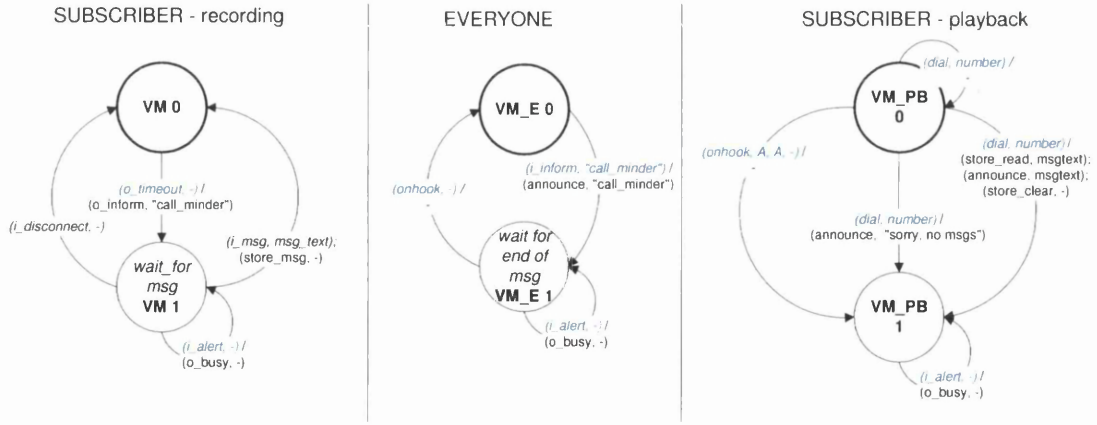

Fig. A.16: Voice Mail Model

Appendix B

Haskell Code Listings

B.1 Module Dependencies

The Haskell model has been developed using modular code. The dependencies of the modules is as follows:

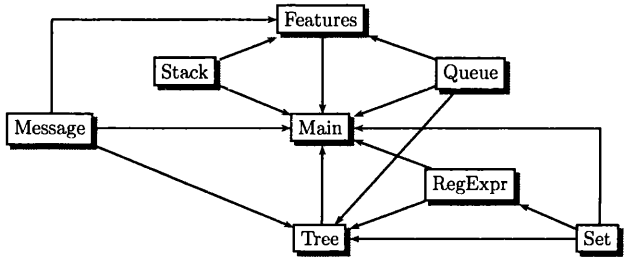

Fig. B.1: Module Dependencies of Haskell Model

The modules `Set`, `Stack` and `Queue` are standard ADT's. `RegExpr` provides the regular expression matching functionality and underlying data type for regular expressions. As these four modules are not of particular interest, their source has not been included here.

The module `Tree` is not a standard ADT, as it includes the pruning and extraction functions, as well as the on-the-fly insertion.

The modules `Message` and `Features` provide the definition of the messages and features that are available in the system. To add new features or messages changes to these two files are required.

The module `Main` contains the feature manager implementation and provides a function `system` that forms the main function.

B.2 Tree.hs

```
1   -- ADT: Tree
2
3   module Tree
4     (Tree, Tree (..), Augtree, OTFtree,
5      isEmptyTree, insertTree, movemarker, traverseTree, traverseAugtree,
6      extractTree, pruneTree, extractdupTree,
7      otftree2augtree, reminsmarker, compacttree, insertOTFTree, insertdelmarker
8     ) where
9
10  import Set
11  import Queue
12  import RegExpr
13
14  -- an augmented tree: stores messages and the numbers of features that send them
15  type Augtree a = Tree (a, Set Int)
16
17  -- tree for on the fly : augtree plus next dfa states
18  type OTFtree a = Tree (a, Set Int, Set Int)
19
20  -- we want equality and a way to show tree
21  instance Eq a => Eq (Tree a) where (==) = eqTree
22  instance Show a => Show (Tree a) where show = showTree
23
24  data Tree a = EmptyTree | InsMarker | DelMarker | Mk a [Tree a]
25
26  isEmptyTree :: Tree a -> Bool
27  isEmptyTree EmptyTree = True
28  isEmptyTree InsMarker = False
29  isEmptyTree DelMarker = False
30  isEmptyTree (Mk x ys) = False
31
32  isInsMarker :: Tree a -> Bool
33  isInsMarker InsMarker = True
34  isInsMarker DelMarker = False
35  isInsMarker EmptyTree = False
36  isInsMarker (Mk x ys) = False
37
38  isDelMarker :: Tree a -> Bool
39  isDelMarker DelMarker = True
40  isDelMarker InsMarker = False
41  isDelMarker EmptyTree = False
42  isDelMarker (Mk x ys) = False
43
44  insertTree :: Tree a -> a -> Tree a
45  insertTree EmptyTree n = Mk n [InsMarker]
46  insertTree DelMarker n = error "insertTree:_trying_to_insert_over_DelMarker"
47  insertTree InsMarker n = Mk n [InsMarker]
48  insertTree (Mk n (x:xs)) y | isInsMarker x = Mk n ([Mk y [InsMarker]] ++ xs)
49                             | otherwise = Mk n ([insertTree x y] ++ xs)
50
51  -- move the insert marker
52  movemarker :: Tree a -> Tree a
53  movemarker EmptyTree = EmptyTree
54  movemarker DelMarker = DelMarker
55  movemarker InsMarker = InsMarker
56  movemarker (Mk n (x:xs)) | isInsMarker x = Mk n (x:xs)
57                           | length mm2x > 1 = Mk n (mm2x ++ xs)
58                           | otherwise = Mk n ([movemarker x] ++ xs)
59                             where mm2x = mm2 x
60
61  mm2 :: Tree a -> [Tree a]
62  mm2 (Mk n (x:xs)) | isInsMarker x && length xs == 0 = InsMarker : [(Mk n [])]
63                    | isInsMarker x && length xs > 0 = InsMarker : [(Mk n xs)]
64                    | otherwise = [(Mk n xs)]
65
66  insertdelmarker :: Tree a -> Tree a
67  insertdelmarker t = movemarker (insertdelmarker1 t)
68
69  insertdelmarker1 :: Tree a -> Tree a
70  insertdelmarker1 EmptyTree = EmptyTree
71  insertdelmarker1 DelMarker = DelMarker
72  insertdelmarker1 InsMarker = InsMarker
73  insertdelmarker1 (Mk n (x:xs)) | isInsMarker x = Mk n (x:DelMarker:xs)
74                                 | otherwise = Mk n ([insertdelmarker1 x] ++ xs)
75
76  -- remove the insertion marker, it's in the leftmost tree
77  reminsmarker :: Tree a -> Tree a
78  reminsmarker InsMarker = EmptyTree
79  reminsmarker DelMarker = DelMarker
80  reminsmarker EmptyTree = EmptyTree
81  reminsmarker (Mk n (x:xs)) | isInsMarker x = Mk n (xs)
82                             | otherwise = reminsmarker x
83
```

```
84  traverseTree :: Tree a -> [a]
85  traverseTree EmptyTree = []
86  traverseTree InsMarker = []
87  traverseTree DelMarker = []
88  traverseTree (Mk n []) = [n]
89  traverseTree (Mk n xs) = n: (concat (map traverseTree xs))
90
91  traverseAugtree :: Augtree a -> [(a, Set Int)]
92  traverseAugtree EmptyTree = []
93  traverseAugtree InsMarker = []
94  traverseAugtree DelMarker = []
95  traverseAugtree (Mk n []) = [n]
96  traverseAugtree (Mk n xs) = n: (concat (map traverseAugtree xs))
97
98  otftree2augtree :: Ord a => OTFtree a -> Augtree a
99  otftree2augtree t = otftree2augtree2 (otfcleanup t)
100
101 otftree2augtree2 :: Ord a => OTFtree a -> Augtree a
102 otftree2augtree2 EmptyTree = EmptyTree
103 otftree2augtree2 InsMarker = InsMarker
104 otftree2augtree2 DelMarker = DelMarker
105 otftree2augtree2 (Mk (m,f,s) []) = Mk (m,f) []
106 otftree2augtree2 (Mk (m,f,s) xs) = Mk (m,f) (map otftree2augtree2 xs)
107
108 otfcleanup :: Ord a => OTFtree a -> OTFtree a
109 otfcleanup EmptyTree = EmptyTree
110 otfcleanup InsMarker = InsMarker
111 otfcleanup DelMarker = DelMarker
112 otfcleanup (Mk n xs) | onlymarkers temp = DelMarker
113                      | otherwise = (Mk n (compact temp))
114                      where temp = map otfcleanup xs
115
116 -- prune by re, then extract duplicates again ... might have introduced new ones!
117 pruneTree :: Ord a => DFA a -> Augtree a -> Augtree a
118 pruneTree re t = prunetraverse re t
119
120 extractdupTree :: Ord a => Augtree a -> Augtree a
121 extractdupTree EmptyTree = EmptyTree
122 extractdupTree InsMarker = EmptyTree
123 extractdupTree DelMarker = EmptyTree
124 extractdupTree (Mk n xs) = Mk n (compact (listremeq newxs))
125                            where newxs = map extractdupTree xs
126
127 listremeq :: Eq a => [a] -> [a]
128 listremeq [] = []
129 listremeq [x] = [x]
130 listremeq (x:xs) | elem x xs = listremeq xs
131                  | otherwise = x:(listremeq xs)
132
133 extractTree :: Ord a => Augtree a -> Augtree a
134 extractTree t = expickone (exmostsat t)
135
136 expriobynumber :: Ord a => Augtree a -> Augtree a
137 expriobynumber t = exmostsat2 t1 (exbyn t1)
138                    where
139                        t1 = compacttree t
140
141 expriobyweight :: Ord a => Augtree a -> [Int] -> Augtree a
142 expriobyweight t w = exmostsat2 t1 (exbyw t1 w)
143                      where
144                          t1 = compacttree t
145
146 exbyn :: Ord a => Augtree a -> [Int]
147 exbyn t = map maxsmallest (fsontrace emptySet t [])
148
149 exbyw :: Ord a => Augtree a -> [Int] -> [Int]
150 exbyw t w = map (calcweight w) (fsontrace emptySet t [])
151
152 maxsmallest :: Set Int -> Int
153 maxsmallest s = 100 - (minl (set2List s))
154
155 calcweight :: [ Int] -> Set Int -> Int
156 calcweight w s = sum (calcw w (set2List s))
157
158 calcw :: [ Int] -> [Int] -> [Int]
159 calcw w [] = []
160 calcw w (x:xs) = (w!!(x)):(calcw w xs)
161
162 minl :: [ Int] -> Int
163 minl [] = 0
164 minl [x] = x
165 minl (x:xs) | x /= 0 = min x (minl xs)
166             | otherwise = minl xs        -- input messages have 0, exclude them
167                                          -- as they don't originate from a feature
168
169 fsontrace :: Ord a => Set Int -> Augtree a -> [Set Int] -> [Set Int]
```

```
fsontrace s EmptyTree l = l ++ [s]
fsontrace s InsMarker l = l ++ [s]
fsontrace s DelMarker l = l ++ [s]
fsontrace s (Mk (n,f) []) l = l ++ [(unionSet s f)]
fsontrace s (Mk (n,f) xs) l = l ++ makelist (unionSet s f) xs
                                  where
                                      makelist s [] = [s]
                                      makelist s [x] = fsontrace s x []
                                      makelist s (x:xs) = (fsontrace s x []) ++ makelist s xs


-- remove del markers and emptytree values from tree (appart from root, that can be emptytree)
compacttree :: Ord a => Augtree a -> Augtree a
compacttree InsMarker = InsMarker
compacttree DelMarker = EmptyTree
compacttree EmptyTree = EmptyTree
compacttree (Mk n xs) = Mk n (compact xs)

exmostsat :: Ord a => Augtree a -> Augtree a
exmostsat t = exmostsat2 t1 (exmostsat1 emptySet t1 [])
              where
                        t1 = compacttree t

exmostsat1 :: Ord a => Set Int -> Augtree a -> [Int] -> [Int]
exmostsat1 s EmptyTree l = l ++ [card s]
exmostsat1 s InsMarker l = l ++ [card s]
exmostsat1 s DelMarker l = l ++ [card s]
exmostsat1 s (Mk (n,f) []) l = l ++ [card (unionSet s f)]
exmostsat1 s (Mk (n,f) xs) l = l ++ makelist (unionSet s f) xs
                                  where
                                      makelist s [] = [card s]
                                      makelist s [x] = exmostsat1 s x []
                                      makelist s (x:xs) = (exmostsat1 s x []) ++ makelist s xs

maxl :: [Int] -> Int
maxl [] = 0
maxl [x] = x
maxl (x:xs) = max x (maxl xs)

 splitlist   :: [ Int] -> [Int] -> [[Int]]
 splitlist  xs  [] = [[]]
 splitlist  xs [y] = [take y xs]
 splitlist  xs (y:ys) = (take y xs):( splitlist  (drop y xs) ys)

exmostsat2 :: Ord a => Augtree a -> [Int] -> Augtree a
exmostsat2 t l | [] == temp = EmptyTree
               | otherwise = head temp
               where
                   temp = exms (maxl l) [t] [l]

exms :: Ord a => Int -> [Augtree a] -> [[Int]] -> [Augtree a]
exms lmax [] _ = []
exms lmax [InsMarker] [y] = [InsMarker]
exms lmax [DelMarker] [y] = [DelMarker]
exms lmax [EmptyTree] [y] = [EmptyTree]
exms lmax [Mk n xs] [y] | ( filter  (ismax lmax) y) == [] = [DelMarker]
                        | otherwise = [Mk n (compact (exms lmax xs (splitlist y
                                                                    (map leavesint xs))))]
exms lmax (x:xs) (y:ys) = compact((head (exms lmax [x] [y])):(exms lmax xs ys))

ismax :: Int -> Int -> Bool
ismax a b = a == b

 leavesint  :: Ord a => Augtree a -> Int
 leavesint  EmptyTree = 0
 leavesint  InsMarker = 0
 leavesint  DelMarker = 0
 leavesint  (Mk n []) = 1
 leavesint  (Mk n xs) = sum (map leavesint xs)

-- pick leftmost branch
expickone :: Ord a => Augtree a -> Augtree a
expickone t = expickone1 (compacttree t)

expickone1 :: Ord a => Augtree a -> Augtree a
expickone1 EmptyTree = EmptyTree
expickone1 InsMarker = EmptyTree
expickone1 DelMarker = EmptyTree
expickone1 (Mk n []) = Mk n []
expickone1 (Mk n [x]) = Mk n [expickone1 x]
expickone1 (Mk n (x:xs)) = Mk n [expickone1 x]

prunetraverse :: Ord a => DFA a -> Augtree a -> Augtree a
prunetraverse dfa EmptyTree = EmptyTree
prunetraverse dfa InsMarker = InsMarker
prunetraverse dfa DelMarker = DelMarker
```

```
256  prunetraverse dfa (Mk n xs) | onlymarkers pl = DelMarker
257                              | otherwise = useDFA2prune dfa (Mk n (compact pl))
258                              where pl = map (prunetraverse dfa) xs
259
260  onlymarkers :: Ord a => [Tree a] -> Bool
261  onlymarkers [] = False
262  onlymarkers [x] | isDelMarker x = True
263                  | otherwise = False
264  onlymarkers (x:xs) | not (isDelMarker x) = False
265                     | otherwise = onlymarkers xs
266
267  compact :: Ord a => [Tree a] -> [Tree a]
268  compact [] = []
269  compact [x] | isEmptyTree x = []
270              | isDelMarker x = []
271              | otherwise = [x] -- keep proper nodes and insertion markers
272  compact (x:xs) | isEmptyTree x = compact xs
273                 | isDelMarker x = compact xs
274                 | otherwise = x:(compact xs) -- keep proper nodes and insertion markers
275
276  useDFA2prune :: Ord a => DFA a -> Augtree a -> Augtree a
277  useDFA2prune (s, a, tr) input | isMemberSet a s = DelMarker   -- startstate is accepting,
278                                                               -- matches, remove tree & mark
279                                | otherwise = useDFA2prune1 a tr s input -- need to look further ...
280
281  useDFA2prune1 :: Ord a => Set Int -> TTBL Int a -> Int -> Augtree a -> Augtree a
282  useDFA2prune1 a tr s EmptyTree = EmptyTree
283  useDFA2prune1 a tr s InsMarker = InsMarker
284  useDFA2prune1 a tr s DelMarker = DelMarker
285  useDFA2prune1 a tr s (Mk n xs) | isMemberSet a gto -- found accepting state remove node
286                                   = DelMarker
287                                 | gto == -99 -- no transition, hence doesn't match
288                                   = (Mk n xs)
289                                 | onlymarkers pl
290                                   = DelMarker
291                                 | otherwise = Mk n (compact pl)
292                                   where gto = goto s tr (fst n)
293                                         pl = map (useDFA2prune1 a tr gto) xs
294
295  insertOTFTree :: Ord a => OTFtree a -> (a, Set Int) -> DFA a -> (Bool, OTFtree a)
296  insertOTFTree EmptyTree (m, f) (s, a, tr) | isMemberSet a s || isMemberSet a gto -- found match:
297                                                                                   -- startstate or gto s m is final
298                                            = (False, DelMarker)
299                                          | otherwise
300                                            = (True, Mk (m, f, insertSet emptySet gto) [InsMarker])
301                                            where gto = goto s tr m
302  insertOTFTree DelMarker n dfa = error "insertOTFTree:_trying_to_insert_over_DelMarker"
303  insertOTFTree InsMarker (m, f) (s, a, tr ) | isMemberSet a s || isMemberSet a gto -- found match:
304                                                                                   -- startstate or gto s m is final
305                                             = (False, DelMarker)
306                                           | otherwise
307                                             = (True, Mk (m, f, insertSet emptySet gto) [InsMarker])
308                                             where gto = goto s tr m
309  insertOTFTree (Mk (m, f, st) (x:xs)) (n, g) (s, a, tr ) | isInsMarker x && (not acc)
310                                                             -- have found insertion point and no match
311                                                           = (True, Mk (m, f, st)
312                                                                       ([Mk (n, g, unionSet st gto) [InsMarker]] ++ xs))
313                                                           | isInsMarker x && acc
314                                                             -- have found insertion point and match
315                                                           = (False, Mk (m, f, st ) ([InsMarker, DelMarker] ++ xs))
316                                                           | otherwise -- haven't found insertion point
317                                                           = (fst ins , Mk (m, f, st ) ([snd(ins)] ++ xs))
318                                                           where ins = insertOTFTree x (n, g) (s, a, tr)
319                                                                 gto = allmoves (s, a, tr ) n (set2List st)
320                                                                 acc = accepting gto a
321
322  allmoves :: Ord a => DFA a -> a -> [Int] -> (Set Int)
323  allmoves (s, a, tr) m [] = emptySet
324  allmoves (s, a, tr) m (x:xs) = insertSet (insertSet (allmoves (s, a, tr) m xs) (goto x tr m)) (goto s tr m)
325
326  accepting :: (Set Int) -> (Set Int) -> Bool
327  accepting x fin = (setInter fin x /= emptySet)
328
329  eqTree :: Eq a => Tree a -> Tree a -> Bool
330  eqTree (Mk x xs) (Mk y ys) = x == y && xs == ys
331  eqTree EmptyTree EmptyTree = True
332  eqTree InsMarker InsMarker = True
333  eqTree DelMarker DelMarker = True
334  eqTree _ _ = False
335
336  showTree :: Show a => Tree a -> String
337  showTree EmptyTree = "empty"
338  showTree InsMarker = "insmarker"
339  showTree DelMarker = "delmarker"
340  showTree (Mk x xs) = show x ++ "[" ++ sh xs ++ "]"
341
```

```
342  sh :: Show a => [a] -> String
343  sh [] = ""
344  sh (x:[]) = show x
345  sh (x:xs) = show x ++ "␣" ++ sh xs
```

B.3 Message.hs

```
1   --
2   -- Module for Message, Event and Argument definitions
3   --
4
5   module Message
6     (Message, Event(..), Arg(..), Io (..)),
7      p_ioasp, p_event, p_argument, p_dest, is_fmsg, is_tcmsg
8   where
9
10  --
11  -- data type for events (enumeration of all possible values)
12  --
13  data Event = Onhook | Offhook | Dial | Flash
14               | Announce | Display | Ringtone | Busytone | Dialtone | Timeouttone
15               | Disconnecttone | Connect | Stopalert | Alert | Cwtone | Storemsg
16               | Storeread | Storeclear
17               | O_notify | O_inform | O_alert | O_free | O_busy | O_stopalert
18               | O_connect | O_disconnect | O_timeout | O_msg | O_request
19               | I_notify | I_inform | I_alert | I_free | I_busy | I_stopalert
20               | I_connect | I_disconnect | I_timeout | I_msg | I_request
21               | Billing_offhook | Billing_onhook | Billing_forwarded | Billing_reverse
22               | Billing_split | Billing_start | Billing_stop
23               | Rollback_transaction | Commit_transaction | Start_transaction | Abort_transaction
24              deriving(Eq, Ord, Show)
25
26  --
27  -- data type for arguments (enumeration of all possible values)
28  --
29  data Arg = Nil | Askpin | Wrongpin | Time | Screened | Splitfactor | Callerid | Barred
30           | Nomsg | Msgtext | Callminder | Id | Ringback | Fastalert | Cwhold | Originator
31          deriving(Eq, Ord, Show)
32
33  --
34  -- data type for ioaspect (enumeration of all possible values)
35  --
36  data Io = Rcv | Snd
37          deriving(Eq, Ord, Show)
38
39  --
40  -- data type for message (a qudruple, the projection functions, and some tests)
41  --
42  type Message = (Io, Event, Arg, Int)
43
44  p_ioasp :: Message -> Io
45  p_ioasp(i,e,a,d) = i
46
47  p_event :: Message -> Event
48  p_event(i,e,a,d) = e
49
50  p_argument :: Message -> Arg
51  p_argument(i,e,a,d) = a
52
53  p_dest :: Message -> Int
54  p_dest(i,e,a,d) = d
55
56  is_fmsg :: Message -> Bool -- is a feature message
57  is_fmsg(i,e,a,d) = not(is_tcmsg(i,e,a,d))
58
59  is_tcmsg :: Message -> Bool -- is a transaction control message
60  is_tcmsg(i,e,a,d) = (e == Rollback_transaction) || (e == Commit_transaction)
61                      || (e == Start_transaction) || (e == Abort_transaction)
```

B.4 Features.hs

```
1   --
2   -- Module providing feature definitions and the cocoon
3   --
4
5   module Features
6     (cocoon, -- the cocoon
7      CState -- a state of a cocoon
8      ) where
```

```
 9
10  import Message
11  import Queue
12  import Stack
13
14  --
15  -- State of the cocoon
16  --
17  type CState = (Int, Stack Int, Bool, Int) -- current state, rollback stack, playback flag and feature id
18
19  --
20  -- the cocoon
21  --
22  -- the cocoon's parameters are the features current state, its current rollback stack
23  -- and also which feature it represents and the connection number and the trigger message and the playbackstatus
24  -- a message queue is returned representing the features responses, the new current state and
25  -- the rollback stack, and the playbackstatus
26  -- the latter three are required for reinstantiation (done by recursion in Lotos)
27  cocoon :: Int -> Stack Int -> Int -> Int -> Message -> Bool -> (Queue Message, CState)
28  cocoon cs rb fid co m f | is_tcmsg m && (p_dest(m) == co || p_dest(m) == 0) && f == False -- not a playback cocoon
29                           = transcontrol cs rb m fid
30                         | p_event(m) == Commit_transaction && (p_dest(m) == fid || p_dest(m) == 0) && f == True
31                           -- we are a playback cocoon but playback has just finished
32                           = (emptyQueue, (cs, rb, False, fid))
33                         | is_fmsg m && (p_dest(m) == co || p_dest(m) == 0) && f == False -- not a playback cocoon
34                           = (snd(featbehave cs m fid), (fst(featbehave cs m fid), rb, f, fid))
35                         | is_fmsg m && (p_dest(m) == co || p_dest(m) == 0) && f == True -- playback cocoon
36                           = (emptyQueue, (fst(featbehave cs m fid), rb, f, fid))
37                         | otherwise = (emptyQueue, (cs, rb, f, fid))
38
39  transcontrol :: Int -> Stack Int -> Message -> Int -> (Queue Message, CState)
40  transcontrol cs rb m n | p_event(m) == Abort_transaction = (emptyQueue, (bottom rb, emptyStack, False, n))
41                         | p_event(m) == Commit_transaction = (emptyQueue, (cs, emptyStack, True, n))
42                         | p_event(m) == Rollback_transaction = (emptyQueue, (top rb, pop rb, False, n))
43                         | p_event(m) == Start_transaction = (emptyQueue, (cs, push rb cs, False, n))
44
45  -- the feature behaviour
46  -- the second Int defines the feature
47  -- the resonse from the features is depending on the State and the Message
48  featbehave :: Int -> Message -> Int -> (Int, Queue Message)
49  featbehave s m f | f == 1   = tl s m   -- the teenline feature
50                   | f == 2   = cfb s m  -- call forwarding busy
51                   | f == 3   = tcs s m  -- terminating call screening
52                   | f == 4   = rc s m   -- reverse charging
53                   | f == 5   = sb s m   -- split billing
54                   | f == 6   = bcs s m  -- basic call
55                   | f == 7   = cnd s m  -- calling number display
56                   | f == 8   = cndb s m -- calling number delivery blocking
57                   | f == 9   = ct s m   -- call transfer
58                   | f == 10  = cw s m   -- call waiting
59                   | f == 11  = gr s m   -- group ringing
60                   | f == 12  = rb s m   -- ringback
61                   | f == 13  = twc s m  -- three way calling
62                   | f == 14  = vmr s m  -- voice mail - recording
63                   | f == 15  = vmpb s m -- voice mail - playback
64                   | otherwise = error "feature_unknown"
65
66  -- the teenline feature
67  tl :: Int -> Message -> (Int, Queue Message)
68  tl n m | n == 0 && p_event(m) == Offhook
69           = (1, enqueue emptyQueue (Snd, Announce, Askpin, 0))
70         | n == 0 && p_event(m) /= Offhook
71           = (0, emptyQueue)
72         | n == 1 && p_event(m) == Onhook
73           = (0, emptyQueue)
74         | n == 1 && p_event(m) == Dial
75           = (2, enqueue emptyQueue (Snd, Announce, Wrongpin, 0))
76         | n == 1 && p_event(m) == I_alert
77           = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
78         | n == 1 && (p_event(m) /= Onhook || p_event(m) /= Dial || p_event(m) /= I_alert)
79           = (1, emptyQueue)
80         | n == 2 && p_event(m) == Onhook
81           = (0, emptyQueue)
82         | n == 2 && p_event(m) == I_alert
83           = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
84         | n == 2 && (p_event(m) /= Onhook || p_event(m) /= I_alert)
85           = (2, emptyQueue)
86
87  -- call forwarding busy
88  cfb :: Int -> Message -> (Int, Queue Message)
89  cfb n m | n == 0 && p_event(m) == O_busy
90            = (0, enqueue (enqueue (enqueue emptyQueue (Snd, O_alert, Nil, 0))
91                                                  (Snd, Billing_forwarded, Nil, 0))
92                                                  (Snd, O_notify, Nil, 0))
93          | n == 0 && p_event(m) /= O_busy
94            = (0, emptyQueue)
```

```
95
96  -- terminating call screening
97  tcs :: Int -> Message -> (Int, Queue Message)
98  tcs n m | n == 0 && p_event(m) == I_alert
99            = (1, enqueue emptyQueue (Snd, O_request, Callerid, 0))
100         | n == 0 && p_event(m) /= I_alert
101           = (0, emptyQueue)
102         | n == 1 && p_event(m) == I_inform
103           = (0, enqueue emptyQueue (Snd, Announce, Screened, 0))
104         | n == 1 && p_event(m) == I_alert
105           = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
106         | n == 1 && (p_event(m) /= I_alert || p_event(m) /= I_inform)
107           = (0, emptyQueue)
108
109 -- reverse charging
110 rc :: Int -> Message -> (Int, Queue Message)
111 rc n m | n == 0 && p_event(m) == Dial
112          = (0, enqueue emptyQueue (Snd, Billing_reverse, Nil, 0))
113        | n == 0 && p_event(m) /= Dial
114          = (0, emptyQueue)
115
116 -- split billing
117 sb :: Int -> Message -> (Int, Queue Message)
118 sb n m | n == 0 && p_event(m) == Dial
119          = (0, enqueue emptyQueue (Snd, Billing_split, Splitfactor , 0))
120        | n == 0 && p_event(m) /= Dial
121          = (0, emptyQueue)
122
123 -- basic call
124 bcs :: Int -> Message -> (Int, Queue Message)
125 bcs n m | n == 0 && p_event(m) == Offhook
126           = (1, enqueue (enqueue emptyQueue (Snd, Billing_offhook, Time, 0))
127                         (Snd, Dialtone, Nil , 0))
128         | n == 0 && p_event(m) == I_alert
129           = (6, enqueue (enqueue emptyQueue (Snd, O_free, Nil, 0))
130                         (Snd, Alert, Nil , 0))
131         | n == 0 && (p_event(m) /= Offhook || p_event(m) /= I_alert)
132           = (0, emptyQueue)
133         | n == 1 && p_event(m) == Dial
134           = (2, enqueue emptyQueue (Snd, O_alert, Nil, 0))
135         | n == 1 && p_event(m) == Onhook
136           = (0, enqueue emptyQueue (Snd, Billing_onhook, Time, 0))
137         | n == 1 && p_event(m) == I_alert
138           = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
139         | n == 1 && (p_event(m) /= Dial || p_event(m) /= Onhook || p_event(m) /= I_alert)
140           = (1, emptyQueue)
141         | n == 2 && p_event(m) == Onhook
142           = (0, enqueue (enqueue emptyQueue (Snd, Billing_onhook, Time, 0))
143                         (Snd, O_stopalert, Nil , 0))
144         | n == 2 && p_event(m) == I_busy
145           = (5, enqueue emptyQueue (Snd, Busytone, Nil, 0))
146         | n == 2 && p_event(m) == I_free
147           = (3, enqueue emptyQueue (Snd, Ringtone, Nil, 0))
148         | n == 2 && p_event(m) == I_alert
149           = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
150         | n == 2 && (p_event(m) /= I_busy || p_event(m) /= Onhook ||
151                      p_event(m) /= I_free || p_event(m) /= I_alert)
152           = (2, emptyQueue)
153         | n == 3 && p_event(m) == Onhook
154           = (0, enqueue (enqueue emptyQueue (Snd, Billing_onhook, Time, 0))
155                         (Snd, O_stopalert, Nil , 0))
156         | n == 3 && p_event(m) == I_timeout
157           = (5, enqueue emptyQueue (Snd, Timeouttone, Nil, 0))
158         | n == 3 && p_event(m) == I_connect
159           = (4, enqueue (enqueue emptyQueue (Snd, Billing_start, Time, 0))
160                         (Snd, Connect, Nil, 0))
161         | n == 3 && p_event(m) == I_alert
162           = (3, enqueue emptyQueue (Snd, O_busy, Nil, 0))
163         | n == 3 && (p_event(m) /= I_timeout || p_event(m) /= Onhook ||
164                      p_event(m) /= I_connect || p_event(m) /= I_alert)
165           = (3, emptyQueue)
166         | n == 4 && p_event(m) == Onhook
167           = (0,  enqueue (enqueue (enqueue emptyQueue (Snd, Billing_stop, Time, 0))
168                                   (Snd, Billing_onhook, Nil , 0))
169                                   (Snd, O_disconnect, Nil, 0))
170         | n == 4 && p_event(m) == I_disconnect
171           = (4, enqueue emptyQueue (Snd, Disconnecttone, Nil, 0))
172         | n == 4 && p_event(m) == I_alert
173           = (4, enqueue emptyQueue (Snd, O_busy, Nil, 0))
174         | n == 4 && (p_event(m) /= Onhook || p_event(m) /= I_disconnect || p_event(m) /= I_alert)
175           = (4, emptyQueue)
176         | n == 5 && p_event(m) == Onhook
177           = (0, enqueue emptyQueue (Snd, Billing_onhook, Time, 0))
178         | n == 5 && p_event(m) == I_alert
179           = (5, enqueue emptyQueue (Snd, O_busy, Nil, 0))
180         | n == 5 && (p_event(m) /= Onhook || p_event(m) /= I_alert)
```

```
181             = (5, emptyQueue)
182           | n == 6 && p_event(m) == I_stopalert
183             = (0,  enqueue emptyQueue (Snd, Stopalert, Nil, 0))
184           | n == 6 && p_event(m) == Offhook
185             = (7, enqueue (enqueue emptyQueue (Snd, Billing_offhook, Time, 0))
186                           (Snd, O_connect, Nil, 0))
187           | n == 6 && p_event(m) == I_alert
188             = (6, enqueue emptyQueue (Snd, O_busy, Nil, 0))
189           | n == 6 && (p_event(m) /= I_stopalert || p_event(m) /= Offhook || p_event(m) /= I_alert)
190             = (6, emptyQueue)
191           | n == 7 && p_event(m) == Onhook
192             = (0,  enqueue (enqueue (enqueue emptyQueue (Snd, Billing_stop, Time, 0))
193                                     (Snd, Billing_onhook, Time, 0))
194                                     (Snd, O_disconnect, Nil, 0))
195           | n == 7 && p_event(m) == I_disconnect
196             = (8, enqueue emptyQueue (Snd, Disconnecttone, Nil, 0))
197           | n == 7 && p_event(m) == I_alert
198             = (7, enqueue emptyQueue (Snd, O_busy, Nil, 0))
199           | n == 7 && (p_event(m) /= Onhook || p_event(m) /= I_disconnect || p_event(m) /= I_alert)
200             = (7, emptyQueue)
201           | n == 8 && p_event(m) == Onhook
202             = (0, enqueue emptyQueue (Snd, Billing_onhook, Time, 0))
203           | n == 8 && p_event(m) == I_alert
204             = (8, enqueue emptyQueue (Snd, O_busy, Nil, 0))
205           | n == 8 && (p_event(m) /= Onhook || p_event(m) /= I_alert)
206             = (8, emptyQueue)
207
208   -- calling number display
209   cnd :: Int -> Message -> (Int, Queue Message)
210   cnd n m | n == 0 && p_event(m) == I_alert
211             = (1, enqueue emptyQueue (Snd, O_request, Callerid, 0))
212           | n == 0 && p_event(m) /= I_alert
213             = (0, emptyQueue)
214           | n == 1 && p_event(m) == I_alert
215             = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
216           | n == 1 && p_event(m) == I_inform
217             = (0, enqueue emptyQueue (Snd, Display, Callerid, 0))
218           | n == 1 && (p_event(m) /= I_alert || p_event(m) /= I_inform)
219             = (1, emptyQueue)
220
221   -- calling number delivery blocking
222   cndb :: Int -> Message -> (Int, Queue Message)
223   cndb n m | n == 0 && p_event(m) == I_request
224              = (0, enqueue emptyQueue (Snd, O_inform, Barred, 0))
225            | n == 0 && p_event(m) /= I_request
226              = (0, emptyQueue)
227
228   -- call transfer
229   ct :: Int -> Message -> (Int, Queue Message)
230   ct n m | n == 0 && p_event(m) == I_connect
231            = (1, emptyQueue)
232          | n == 0 && p_event(m) == O_connect
233            = (7, emptyQueue)
234          | n == 0 && (p_event(m) /= I_connect || p_event(m) /= O_connect)
235            = (0, emptyQueue)
236          | n == 1 && p_event(m) == I_disconnect
237            = (0, emptyQueue)
238          | n == 1 && p_event(m) == Onhook
239            = (0, emptyQueue)
240          | n == 1 && p_event(m) == I_alert
241            = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
242          | n == 1 && p_event(m) == Flash
243            = (2, enqueue emptyQueue (Snd, Dialtone, Nil, 0))
244          | n == 1 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
245                       p_event(m) /= I_alert || p_event(m) /= Flash)
246            = (1, emptyQueue)
247          | n == 2 && p_event(m) == I_disconnect
248            = (2, emptyQueue)
249          | n == 2 && p_event(m) == Flash
250            = (1, emptyQueue)
251          | n == 2 && p_event(m) == I_alert
252            = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
253          | n == 2 && p_event(m) == Dial
254            = (3, enqueue emptyQueue (Snd, O_alert, Nil, 0))
255          | n == 2 && (p_event(m) /= I_disconnect || p_event(m) /= Dial ||
256                       p_event(m) /= I_alert || p_event(m) /= Flash)
257            = (2, emptyQueue)
258          | n == 3 && p_event(m) == I_disconnect
259            = (3, emptyQueue)
260          | n == 3 && p_event(m) == I_alert
261            = (3, enqueue emptyQueue (Snd, O_busy, Nil, 0))
262          | n == 3 && p_event(m) == Flash
263            = (1, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
264          | n == 3 && p_event(m) == I_free
265            = (5, enqueue emptyQueue (Snd, Ringtone, Nil, 0))
266          | n == 3 && p_event(m) == I_busy
```

```
267           = (4, enqueue emptyQueue (Snd, Busytone, Nil, 0))
268         | n == 3 && (p_event(m) /= I_disconnect || p_event(m) /= I_busy ||
269                      p_event(m) /= I_alert || p_event(m) /= Flash || p_event(m) /= I_free)
270           = (3, emptyQueue)
271         | n == 4 && p_event(m) == I_disconnect
272           = (0, emptyQueue)
273         | n == 4 && p_event(m) == Flash
274           = (1, emptyQueue)
275         | n == 4 && p_event(m) == I_alert
276           = (4, enqueue emptyQueue (Snd, O_busy, Nil, 0))
277         | n == 4 && p_event(m) == Onhook
278           = (0, emptyQueue)
279         | n == 4 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
280                      p_event(m) /= I_alert || p_event(m) /= Flash)
281           = (4, emptyQueue)
282         | n == 5 && p_event(m) == I_disconnect
283           = (5, emptyQueue)
284         | n == 5 && p_event(m) == I_alert
285           = (5, enqueue emptyQueue (Snd, O_busy, Nil, 0))
286         | n == 5 && p_event(m) == Flash
287           = (1, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
288         | n == 5 && p_event(m) == Onhook
289           = (0, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
290         | n == 5 && p_event(m) == I_connect
291           = (6, enqueue( enqueue emptyQueue (Snd, Billing_start, Time, 0))
292                         (Snd, Connect, Nil, 0))
293         | n == 5 && p_event(m) == I_timeout
294           = (4, enqueue emptyQueue (Snd, Timeouttone, Nil, 0))
295         | n == 5 && (p_event(m) /= I_disconnect || p_event(m) /= I_alert || p_event(m) /= Flash ||
296                      p_event(m) /= Onhook || p_event(m) /= I_connect || p_event(m) /= I_timeout)
297           = (5, emptyQueue)
298         | n == 6 && p_event(m) == I_disconnect
299           = (6, emptyQueue)
300         | n == 6 && p_event(m) == Onhook
301           = (0, enqueue(enqueue(enqueue emptyQueue (Snd, O_inform, Originator, 0))
302                                                    (Snd, O_notify, Nil, 0))
303                                                    (Snd, O_notify, Nil, 0))
304         | n == 6 && p_event(m) == I_alert
305           = (6, enqueue emptyQueue (Snd, O_busy, Nil, 0))
306         | n == 6 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
307                      p_event(m) /= I_alert)
308           = (6, emptyQueue)
309         | n == 7 && p_event(m) == I_disconnect
310           = (0, emptyQueue)
311         | n == 7 && p_event(m) == Onhook
312           = (0, emptyQueue)
313         | n == 7 && p_event(m) == I_alert
314           = (7, enqueue emptyQueue (Snd, O_busy, Nil, 0))
315         | n == 7 && p_event(m) == Flash
316           = (8, enqueue emptyQueue (Snd, Dialtone, Nil, 0))
317         | n == 7 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
318                      p_event(m) /= I_alert || p_event(m) /= Flash)
319           = (7, emptyQueue)
320         | n == 8 && p_event(m) == I_disconnect
321           = (2, emptyQueue)
322         | n == 8 && p_event(m) == Flash
323           = (1, emptyQueue)
324         | n == 8 && p_event(m) == I_alert
325           = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
326         | n == 8 && p_event(m) == Dial
327           = (3, enqueue emptyQueue (Snd, O_alert, Nil, 0))
328         | n == 8 && (p_event(m) /= I_disconnect || p_event(m) /= Dial ||
329                      p_event(m) /= I_alert || p_event(m) /= Flash)
330           = (2, emptyQueue)
331         | n == 9 && p_event(m) == I_disconnect
332           = (9, emptyQueue)
333         | n == 9 && p_event(m) == I_alert
334           = (9, enqueue emptyQueue (Snd, O_busy, Nil, 0))
335         | n == 9 && p_event(m) == Flash
336           = (7, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
337         | n == 9 && p_event(m) == I_free
338           = (11, enqueue emptyQueue (Snd, Ringtone, Nil, 0))
339         | n == 9 && p_event(m) == I_busy
340           = (10, enqueue emptyQueue (Snd, Busytone, Nil, 0))
341         | n == 9 && (p_event(m) /= I_disconnect || p_event(m) /= I_busy ||
342                      p_event(m) /= I_alert || p_event(m) /= Flash || p_event(m) /= I_free)
343           = (9, emptyQueue)
344         | n == 10 && p_event(m) == I_disconnect
345           = (0, emptyQueue)
346         | n == 10 && p_event(m) == Flash
347           = (7, emptyQueue)
348         | n == 10 && p_event(m) == I_alert
349           = (10, enqueue emptyQueue (Snd, O_busy, Nil, 0))
350         | n == 10 && p_event(m) == Onhook
351           = (0, emptyQueue)
352         | n == 10 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
```

```
353                     p_event(m) /= I_alert || p_event(m) /= Flash)
354           = (10, emptyQueue)
355         | n == 11 && p_event(m) == I_disconnect
356           = (11, emptyQueue)
357         | n == 11 && p_event(m) == I_alert
358           = (11, enqueue emptyQueue (Snd, O_busy, Nil, 0))
359         | n == 11 && p_event(m) == Flash
360           = (7, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
361         | n == 11 && p_event(m) == Onhook
362           = (0, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
363         | n == 11 && p_event(m) == I_connect
364           = (12, enqueue( enqueue emptyQueue (Snd, Billing_start, Time, 0))
365                         (Snd, Connect, Nil, 0))
366         | n == 11 && p_event(m) == I_timeout
367           = (10, enqueue emptyQueue (Snd, Timeouttone, Nil, 0))
368         | n == 11 && (p_event(m) /= I_disconnect || p_event(m) /= I_alert || p_event(m) /= Flash ||
369                     p_event(m) /= Onhook || p_event(m) /= I_connect || p_event(m) /= I_timeout)
370           = (11, emptyQueue)
371         | n == 12 && p_event(m) == I_disconnect
372           = (12, emptyQueue)
373         | n == 12 && p_event(m) == Onhook
374           = (0, enqueue(enqueue emptyQueue (Snd, O_notify, Nil, 0))
375                                         (Snd, O_notify, Nil , 0))
376         | n == 12 && p_event(m) == I_alert
377           = (12, enqueue emptyQueue (Snd, O_busy, Nil, 0))
378         | n == 12 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
379                     p_event(m) /= I_alert)
380           = (12, emptyQueue)
381
382   -- call waiting
383   cw :: Int -> Message -> (Int, Queue Message)
384   cw n m | n == 0 && p_event(m) == I_connect
385           = (1, emptyQueue)
386         | n == 0 && p_event(m) == O_connect
387           = (1, emptyQueue)
388         | n == 0 && (p_event(m) /= I_connect || p_event(m) /= O_connect)
389           = (0, emptyQueue)
390         | n == 1 && p_event(m) == I_disconnect
391           = (0, emptyQueue)
392         | n == 1 && p_event(m) == Onhook
393           = (0, emptyQueue)
394         | n == 1 && p_event(m) == I_alert
395           = (2, enqueue emptyQueue (Snd, O_inform, Cwhold,0))
396         | n == 1 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook || p_event(m) /= I_alert)
397           = (1, emptyQueue)
398         | n == 2 && p_event(m) == I_disconnect
399           = (0, emptyQueue)
400         | n == 2 && p_event(m) == Onhook
401           = (0, enqueue emptyQueue (Snd, O_disconnect, Nil, 0))
402         | n == 2 && p_event(m) == I_alert
403           = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
404         | n == 2 && p_event(m) == Flash
405           = (3, enqueue (enqueue (enqueue (enqueue emptyQueue (Snd, Billing_stop, Time, 0))
406                                                   (Snd, O_inform, Cwhold,0))
407                                                   (Snd, Billing_start , Time, 0))
408                                                   (Snd, O_connect, Nil, 0))
409         | n == 2 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
410                     p_event(m) /= I_alert || p_event(m) /= Flash )
411           = (2, emptyQueue)
412         | n == 3 && p_event(m) == I_disconnect
413           = (0, emptyQueue)
414         | n == 3 && p_event(m) == Onhook
415           = (0, enqueue emptyQueue (Snd, O_disconnect, Nil, 0))
416         | n == 3 && p_event(m) == I_alert
417           = (3, enqueue emptyQueue (Snd, O_busy, Nil, 0))
418         | n == 3 && p_event(m) == Flash
419           = (2, enqueue (enqueue (enqueue (enqueue emptyQueue (Snd, Billing_stop, Time, 0))
420                                                   (Snd, O_inform, Cwhold,0))
421                                                   (Snd, Billing_start , Time, 0))
422                                                   (Snd, O_connect, Nil, 0))
423         | n == 3 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
424                     p_event(m) /= I_alert || p_event(m) /= Flash )
425           = (3, emptyQueue)
426
427   -- group ringing
428   gr :: Int -> Message -> (Int, Queue Message)
429   gr n m | n == 0 && p_event(m) == I_alert
430           = (1, enqueue (enqueue emptyQueue (Snd, O_alert, Nil, 0))
431                                         (Snd, O_alert, Nil , 0))
432         | n == 0 && p_event(m) /= I_alert
433           = (0, emptyQueue)
434         | n == 1 && p_event(m) == I_alert
435           = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
436         | n == 1 && p_event(m) == I_stopalert
437           = (0, enqueue (enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
438                                         (Snd, O_stopalert, Nil , 0))
```

```
439         | n == 1 && p_event(m) == I_busy
440           = (2, emptyQueue)
441         | n == 1 && p_event(m) == I_free
442           = (3, emptyQueue)
443         | n == 1 && (p_event(m) /= I_busy || p_event(m) /= I_free ||
444                      p_event(m) /= I_stopalert || p_event(m) /= I_alert)
445           = (1, emptyQueue)
446         | n == 2 && p_event(m) == I_alert
447           = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
448         | n == 2 && p_event(m) == I_stopalert
449           = (0, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
450         | n == 2 && p_event(m) == I_busy
451           = (0, emptyQueue)
452         | n == 2 && p_event(m) == I_free
453           = (3, emptyQueue)
454         | n == 2 && (p_event(m) /= I_busy || p_event(m) /= I_free ||
455                      p_event(m) /= I_stopalert || p_event(m) /= I_alert)
456           = (2, emptyQueue)
457         | n == 3 && p_event(m) == I_alert
458           = (3, enqueue emptyQueue (Snd, O_busy, Nil, 0))
459         | n == 3 && p_event(m) == I_stopalert
460           = (0, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
461         | n == 3 && p_event(m) == I_free
462           = (3, emptyQueue)
463         | n == 3 && p_event(m) == I_connect
464           = (0, enqueue (enqueue (enqueue (enqueue (enqueue (enqueue
465                        emptyQueue (Snd, O_stopalert, Nil, 0))
466                                   (Snd, Stopalert, Nil, 0))
467                                   (Snd, Billing_forwarded, Nil, 0))
468                                   (Snd, O_free, Nil, 0))
469                                   (Snd, O_notify, Nil, 0))
470                                   (Snd, O_notify, Nil, 0))
471                                   (Snd, O_connect, Nil, 0))
472         | n == 3 && (p_event(m) /= I_connect || p_event(m) /= I_free ||
473                      p_event(m) /= I_stopalert || p_event(m) /= I_alert)
474           = (3, emptyQueue)
475
476   -- ringback
477   rb :: Int -> Message -> (Int, Queue Message)
478   rb n m | n == 0 && p_event(m) == Offhook
479            = (1, emptyQueue)
480          | n == 0 && p_event(m) == I_alert
481            = (0, emptyQueue)
482          | n == 0 && (p_event(m) /= Offhook || p_event(m) /= I_alert)
483            = (0, emptyQueue)
484          | n == 1 && p_event(m) == I_alert
485            = (1, enqueue emptyQueue (Snd, O_inform, Ringback, 0))
486          | n == 1 && p_event(m) == I_stopalert
487            = (2, enqueue emptyQueue (Snd, Alert, Fastalert, 0))
488          | n == 1 && p_event(m) == Onhook
489            = (2, enqueue emptyQueue (Snd, Alert, Fastalert, 0))
490          | n == 1 && p_event(m) == O_timeout
491            = (2, enqueue emptyQueue (Snd, Alert, Fastalert, 0))
492          | n == 1 && (p_event(m) /= I_alert || p_event(m) /= I_stopalert ||
493                       p_event(m) /= Onhook || p_event(m) /= O_timeout)
494            = (1, emptyQueue)
495          | n == 2 && p_event(m) == Offhook
496            = (0, enqueue emptyQueue (Snd, O_alert, Nil, 0))
497          | n == 2 && p_event(m) == I_alert
498            = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
499          | n == 2 && (p_event(m) /= I_alert || p_event(m) /= Offhook)
500            = (2, emptyQueue)
501
502   -- three way calling
503   twc :: Int -> Message -> (Int, Queue Message)
504   twc n m | n == 0 && p_event(m) == I_connect
505             = (1, emptyQueue)
506           | n == 0 && p_event(m) == O_connect
507             = (1, emptyQueue)
508           | n == 0 && (p_event(m) /= I_connect || p_event(m) /= O_connect)
509             = (0, emptyQueue)
510           | n == 1 && p_event(m) == I_disconnect
511             = (0, emptyQueue)
512           | n == 1 && p_event(m) == Onhook
513             = (0, emptyQueue)
514           | n == 1 && p_event(m) == I_alert
515             = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
516           | n == 1 && p_event(m) == Flash
517             = (2, enqueue emptyQueue (Snd, Dialtone, Nil, 0))
518           | n == 1 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
519                        p_event(m) /= I_alert || p_event(m) /= Flash)
520             = (1, emptyQueue)
521           | n == 2 && p_event(m) == I_disconnect
522             = (2, emptyQueue)
523           | n == 2 && p_event(m) == Flash
524             = (1, emptyQueue)
```

```
525           | n == 2 && p_event(m) == I_alert
526             = (2, enqueue emptyQueue (Snd, O_busy, Nil, 0))
527           | n == 2 && p_event(m) == Dial
528             = (3, enqueue emptyQueue (Snd, O_alert, Nil, 0))
529           | n == 2 && (p_event(m) /= I_disconnect || p_event(m) /= Dial ||
530                        p_event(m) /= I_alert || p_event(m) /= Flash)
531             = (2, emptyQueue)
532           | n == 3 && p_event(m) == I_disconnect
533             = (3, emptyQueue)
534           | n == 3 && p_event(m) == Flash
535             = (1, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
536           | n == 3 && p_event(m) == I_alert
537             = (3, enqueue emptyQueue (Snd, O_busy, Nil, 0))
538           | n == 3 && p_event(m) == I_free
539             = (5, enqueue emptyQueue (Snd, Ringtone, Nil, 0))
540           | n == 3 && p_event(m) == I_busy
541             = (4, enqueue emptyQueue (Snd, Busytone, Nil, 0))
542           | n == 3 && (p_event(m) /= I_disconnect || p_event(m) /= I_busy ||
543                        p_event(m) /= I_alert || p_event(m) /= Flash ||
544                        p_event(m) /= I_free)
545             = (3, emptyQueue)
546           | n == 4 && p_event(m) == I_disconnect
547             = (4, emptyQueue)
548           | n == 4 && p_event(m) == Flash
549             = (1, emptyQueue)
550           | n == 4 && p_event(m) == Onhook
551             = (0, emptyQueue)
552           | n == 4 && p_event(m) == I_alert
553             = (4, enqueue emptyQueue (Snd, O_busy, Nil, 0))
554           | n == 4 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
555                        p_event(m) /= I_alert || p_event(m) /= Flash)
556             = (4, emptyQueue)
557           | n == 5 && p_event(m) == I_disconnect
558             = (5, emptyQueue)
559           | n == 5 && p_event(m) == Flash
560             = (1, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
561           | n == 5 && p_event(m) == Onhook
562             = (0, enqueue emptyQueue (Snd, O_stopalert, Nil, 0))
563           | n == 5 && p_event(m) == I_alert
564             = (5, enqueue emptyQueue (Snd, O_busy, Nil, 0))
565           | n == 5 && p_event(m) == I_connect
566             = (6, enqueue (enqueue emptyQueue (Snd, Billing_start, Time, 0))
567                                                (Snd, Connect, Nil, 0))
568           | n == 5 && p_event(m) == I_timeout
569             = (4, enqueue emptyQueue (Snd, Timeouttone, Nil, 0))
570           | n == 5 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
571                        p_event(m) /= I_alert || p_event(m) /= Flash ||
572                        p_event(m) /= I_timeout || p_event(m) /= I_connect)
573             = (5, emptyQueue)
574           | n == 6 && p_event(m) == I_disconnect
575             = (0, emptyQueue)
576           | n == 6 && p_event(m) == Flash
577             = (7, emptyQueue)
578           | n == 6 && p_event(m) == Onhook
579             = (0, enqueue emptyQueue (Snd, O_disconnect, Nil, 0))
580           | n == 6 && p_event(m) == I_alert
581             = (6, enqueue emptyQueue (Snd, O_busy, Nil, 0))
582           | n == 6 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
583                        p_event(m) /= I_alert || p_event(m) /= Flash)
584             = (6, emptyQueue)
585           | n == 7 && p_event(m) == I_disconnect
586             = (0, emptyQueue)
587           | n == 7 && p_event(m) == Onhook
588             = (0, enqueue emptyQueue (Snd, O_disconnect, Nil, 0))
589           | n == 7 && p_event(m) == I_alert
590             = (7, enqueue emptyQueue (Snd, O_busy, Nil, 0))
591           | n == 7 && (p_event(m) /= I_disconnect || p_event(m) /= Onhook ||
592                        p_event(m) /= I_alert)
593             = (7, emptyQueue)
594
595   -- voice mail - recording
596   vmr :: Int -> Message -> (Int, Queue Message)
597   vmr n m | n == 0 && p_event(m) == O_timeout
598             = (1, enqueue emptyQueue (Snd, O_inform, Callminder, 0))
599           | n == 0 && p_event(m) /= O_timeout
600             = (0, emptyQueue)
601           | n == 1 && p_event(m) == I_alert
602             = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
603           | n == 1 && p_event(m) == I_disconnect
604             = (0, emptyQueue)
605           | n == 1 && p_event(m) == I_msg
606             = (0, enqueue emptyQueue (Snd, Storemsg, Nil, 0))
607           | n == 1 && (p_event(m) /= I_alert || p_event(m) /= I_disconnect || p_event(m) /= I_msg)
608             = (1, emptyQueue)
609
610   -- voice mail - playback
```

```
611  vmpb :: Int -> Message -> (Int, Queue Message)
612  vmpb n m | n == 0 && p_event(m) == Dial
613             = (1, enqueue (enqueue (enqueue emptyQueue (Snd, Storeread, Msgtext, 0))
614                                    (Snd, Announce, Msgtext, 0))
615                                    (Snd, Storeclear , Nil , 0))
616           | n == 0 && p_event(m) /= Dial
617             = (0, emptyQueue)
618           | n == 1 && p_event(m) == I_alert
619             = (1, enqueue emptyQueue (Snd, O_busy, Nil, 0))
620           | n == 1 && p_event(m) == Onhook
621             = (0, emptyQueue)
622           | n == 1 && (p_event(m) /= I_alert || p_event(m) /= Onhook)
623             = (1, emptyQueue)
```

B.5 Main.hs

```
1   import Stack
2   import Queue
3   import Set
4   import Message
5   import Tree
6   import Features
7   import RegExpr
8
9   -- type for the State and the stack to save them
10  type State = ([Queue Message], Int)
11  type StateStack = Stack State
12
13  -- the trigger generator
14  -- the parameter determines which message is returned
15  triggergen :: Int -> Message
16  triggergen n | n == 1 = (Rcv, Dial, Nil, 0)
17               | n == 2 = (Rcv, Onhook, Nil, 0)
18               | n == 3 = (Rcv, Offhook, Nil, 0)
19               | n == 4 = (Rcv, Flash, Nil, 0)
20               | n == 5 = (Rcv, I_notify, Nil , 0)
21               | n == 6 = (Rcv, I_inform, Nil, 0)
22               | n == 7 = (Rcv, I_alert, Nil , 0)
23               | n == 8 = (Rcv, I_free, Nil , 0)
24               | n == 9 = (Rcv, I_busy, Nil, 0)
25               | n == 10 = (Rcv, I_timeout, Nil, 0)
26               | n == 11 = (Rcv, I_stopalert, Nil , 0)
27               | n == 12 = (Rcv, I_disconnect, Nil, 0)
28               | n == 13 = (Rcv, I_connect, Nil, 0)
29               | n == 14 = (Rcv, I_msg, Nil, 0)
30               | n == 15 = (Rcv, I_request, Nil, 0)
31               | n == 16 = (Rcv, O_busy, Nil, 0)
32               | n == 17 = (Rcv, O_connect, Nil, 0)
33               | n == 18 = (Rcv, O_timeout, Nil, 0)
34
35  -- otf == True: use on the fly method
36  system :: Bool -> [CState] -> Int -> RegExpr Message -> ([Message], [CState])
37  system otf fs n rules = fman otf fs (triggergen n) (re2dfa rules)
38
39  fman :: Bool -> [CState] -> Message -> DFA Message -> ([Message], [CState])
40  fman otf fs m rules = commit (fm otf fs m rules) fs
41
42  fm :: Bool -> [CState] -> Message -> DFA Message -> Augtree Message
43  fm otf fs m rules | not otf = extract(prune(extractdup(construct fs m)) rules)
44                    | otf     = extract(extractdup(otfconstruct fs m rules))
45
46  construct :: [ CState] -> Message -> Augtree Message
47  construct fs m = reminsmarker(construct1 (snd(toallcocoons fs (Rcv, Start_transaction, Nil, 0))) m)
48
49  otfconstruct :: [ CState] -> Message -> DFA Message -> Augtree Message
50  otfconstruct fs m rules = reminsmarker(otftree2augtree(otfconstruct1
51                                  (snd(toallcocoons fs (Rcv, Start_transaction, Nil , 0))) m rules))
52
53  construct1 :: [ CState] -> Message -> Augtree Message
54  construct1 fs m = feedbackctrl (toallcocoons fs m) (insertTree EmptyTree (m, singSet 0))
55
56  otfconstruct1 :: [ CState] -> Message -> DFA Message -> OTFtree Message
57  otfconstruct1 fs m dfa | fst insotf = (otffeedbackctrl (toallcocoons fs m) (snd insotf) dfa)
58                         | otherwise = EmptyTree -- root violates rules, done -- this really shouldn't happen!
59                           where insotf = insertOTFTree EmptyTree (m, singSet 0) dfa
60
61  feedbackctrl :: ([ Queue Message], [CState]) -> Augtree Message -> Augtree Message
62  feedbackctrl (ms, fs) t = feedbackctrl1 (makeflags (length fs)) ms fs t
63
64  otffeedbackctrl :: ([ Queue Message], [CState]) -> OTFtree Message -> DFA Message -> OTFtree Message
65  otffeedbackctrl (ms, fs) t dfa = otffeedbackctrl1 (makeflags (length fs)) ms fs t dfa
66
67  feedbackctrl1 :: [[ Bool]] -> [Queue Message] -> [CState] -> Augtree Message -> Augtree Message
```

```
feedbackctrl1 [x] ms fs t = fst(feedback x (choosesome x ms) fs t 0 0 emptyStack) -- finish, last case!
feedbackctrl1 (x:xs) ms fs t = (feedbackctrl1 xs ms fs (fst(feedback x (choosesome x ms) fs t 0 0 emptyStack)))
                                        -- need to continue building the same tree

otffeedbackctrl1  :: [[ Bool]] -> [Queue Message] -> [CState] -> OTFtree Message -> DFA Message ->
                                                                                    OTFtree Message
otffeedbackctrl1 [x] ms fs t dfa = fst(otffeedback x (choosesome x ms) fs t 0 0 emptyStack dfa)
                                  -- finish, last case!
otffeedbackctrl1 (x:xs) ms fs t dfa = (otffeedbackctrl1 xs ms fs
                                                (fst(otffeedback x (choosesome x ms) fs t 0 0 emptyStack dfa)) dfa)
                                        -- need to continue building the same tree

-- handle construct prune ...
feedback :: [ Bool] -> [Queue Message] -> [CState] -> (Augtree Message) -> Int -> Int -> StateStack ->
                                                                    (Augtree Message, [CState])
feedback bs ms fs t c d sstk | qe && es && d < 50 = (t, fs) -- finished, just return tree
                             | qe && not (es) && d < 50   -- rollback ...
                               = feedback bs (fst(top sstk)) (snd(toallcocoons fs (Rcv, Rollback_transaction, Nil, 0)))
                                        (movemarker t) (snd(top sstk)) (d - 1) (pop sstk)
                             | not (qe) && c < nonemptyQs ms && d < 50 -- more branches at this level
                               = feedback bs
                                        (appendQs (choosesome bs (snd getm))
                                                (choosesome bs (fst temp)))
                                        (snd temp)
                                        (insertTree t (fst getm))
                                        0 (d + 1) (push sstk (ms, c + 1))
                             | not (qe) && c == nonemptyQs ms && d < 50 -- need to rollback further
                               = feedback bs (emptyQs ms) fs t 0 d sstk
                             | d >= 50 -- we have reached the maximal depth, mark branch and rollback!
                               = feedback bs (fst(top sstk)) (snd(toallcocoons fs (Rcv, Rollback_transaction, Nil, 0)))
                                        (insertdelmarker t) (snd(top sstk)) (d - 1) (pop sstk)
                               where qe = allQsEmpty ms
                                     es = isEmptyStack sstk
                                     startall = snd(toallcocoons fs (Rcv, Start_transaction, Nil, 0))
                                     getfmsg = fst(fst(getMsgFromQ ms c))
                                     getm = getMsgFromQ ms c
                                     temp = toallcocoons startall getfmsg

-- handle construct prune ...
otffeedback :: [ Bool] -> [Queue Message] -> [CState] -> (OTFtree Message) -> Int -> Int -> StateStack ->
                                                                    DFA Message -> (OTFtree Message, [CState])
otffeedback bs ms fs t c d sstk dfa | qe && es && d < 50 = (t, fs) -- finished, just return tree
                                    | qe && not (es) && d < 50   -- rollback ...
                                      = otffeedback bs (fst(top sstk))
                                                (snd(toallcocoons fs (Rcv, Rollback_transaction, Nil, 0)))
                                                (movemarker t) (snd(top sstk)) (d - 1) (pop sstk) dfa
                                    | not (qe) && c < nonemptyQs ms && (fst insotf) && d < 50
                                      -- more branches at this level and insert successfull
                                      = otffeedback bs
                                          (appendQs (choosesome bs (snd getm))
                                                (choosesome bs (fst temp)))
                                          (snd temp)
                                          (snd insotf)
                                          0 (d + 1) (push sstk (ms, c + 1)) dfa
                                    | not (qe) && c < nonemptyQs ms && not (fst insotf) && d < 50
                                      -- more branches at this level but insert failed
                                      = otffeedback bs (fst(top sstk))
                                                (snd(toallcocoons fs (Rcv, Rollback_transaction, Nil, 0)))
                                                (movemarker (snd insotf)) (snd(top sstk)) (d - 1) (pop sstk) dfa
                                    | not (qe) && c == nonemptyQs ms && d < 50 -- need to rollback further
                                      = otffeedback bs (emptyQs ms) fs t 0 d sstk dfa
                                    | d >= 50 -- we have reached the maximal depth, mark branch and rollback!
                                      = otffeedback bs (fst(top sstk))
                                                (snd(toallcocoons fs (Rcv, Rollback_transaction, Nil, 0)))
                                                (insertdelmarker t) (snd(top sstk)) (d - 1) (pop sstk) dfa
                                      where qe = allQsEmpty ms
                                            es = isEmptyStack sstk
                                            startall = snd(toallcocoons fs (Rcv, Start_transaction, Nil, 0))
                                            getfmsg = fst(fst(getMsgFromQ ms c))
                                            getm = getMsgFromQ ms c
                                            temp = toallcocoons startall getfmsg
                                            insotf = insertOTFTree t (fst getm) dfa

choosesome :: [Bool] -> [Queue Message] -> [Queue Message]
choosesome [] [] = []
choosesome (b:bs) (x:xs) | b == True = x:(choosesome bs xs)
                         | b == False = emptyQueue:(choosesome bs xs)

appendQs :: Ord a => [Queue a] -> [Queue a] -> [Queue a]
appendQs [] [] = []
appendQs [q] [r] = [concatQueue q r]
appendQs (q:qs) (r:rs) = (concatQueue q r):(appendQs qs rs)

getMsgFromQ :: [Queue Message] -> Int -> ((Message, Set Int), [Queue Message])
getMsgFromQ q c = getChosenMsg q q c 0
```

```
154  getChosenMsg :: [Queue Message] -> [Queue Message] -> Int -> Int -> ((Message, Set Int), [Queue Message])
155  getChosenMsg (q:qs) r c t | c + 1 == 1 && not eq = ((first q, (insertSet (fst temp) (t + 1))), snd temp)
156                            | c + 1 == 1 && eq = getChosenMsg qs r c (t + 1)
157                            | c + 1 > 1  && not eq = getChosenMsg qs r (c - 1) (t + 1)
158                            | c + 1 > 1  && eq = getChosenMsg qs r c (t + 1)
159                              where eq = isEmptyQueue q
160                                    temp = remFromAllQs 1 emptySet (first q) r
161
162  remFromAllQs :: Int -> Set Int -> Message -> [Queue Message] -> (Set Int, [Queue Message])
163  remFromAllQs i s m [] = (s, [])
164  remFromAllQs i s m [q] | not (isEmptyQueue q) && first q == m = (insertSet s i, [dequeue q])
165                         | otherwise = (s, [q])
166  remFromAllQs i s m (q:qs) | not (isEmptyQueue q) && first q == m
167                              = (unionSet (insertSet s i) (fst temp), [dequeue q] ++ (snd temp))
168                            | otherwise = (unionSet s (fst temp), [q] ++ (snd temp))
169                            where temp = remFromAllQs (i + 1) s m qs
170
171  allQsEmpty :: [Queue a] -> Bool
172  allQsEmpty q | foldr1 (+) (map lengthQueue q) == 0 = True
173               | otherwise = False
174
175  nonemptyQs :: [Queue a] -> Int
176  nonemptyQs q = foldr1 (+) (map bool2int (map isEmptyQueue q))
177
178  bool2int :: Bool -> Int
179  bool2int b | b == True = 0
180             | otherwise = 1
181
182  emptyQs :: [Queue a] -> [Queue a]
183  emptyQs [] = []
184  emptyQs (x:xs) = emptyQueue:(emptyQs xs)
185
186  --generate list of list of booleans, containing all combinations appart from n*false
187  makeflags :: Int -> [[Bool]]
188  makeflags n = makeflags1 ((2^n) - 1) (makeFflags n)
189
190  makeflags1 :: Int -> [Bool] -> [[Bool]]
191  makeflags1 n b | n > 1  = (makeNflag b) : (makeflags1 (n - 1) (makeNflag b))
192                 | n == 1 = [(makeNflag b)]
193
194  makeFflags :: Int -> [Bool]
195  makeFflags n | n == 0 = []
196               | n > 0 = False: (makeFflags (n - 1))
197
198  makeNflag :: [Bool] -> [Bool]
199  makeNflag b = makeNflag1 b (length b) 0
200
201  makeNflag1 :: [Bool] -> Int -> Int -> [Bool]
202  makeNflag1 xs n m | m == n = xs
203                    | m < n && xs!!m == False && m > 0
204                      = (take (m) xs) ++ (True:(drop (m + 1) xs))
205                    | m < n && xs!!m == False && m == 0
206                      = True:(drop 1 xs)
207                    | m < n && xs!!m == True && m > 0
208                      = makeNflag1 ((take (m) xs) ++ (False:(drop (m + 1) xs))) n (m + 1)
209                    | m < n && xs!!m == True && m == 0
210                      = makeNflag1 (False:(drop 1 xs)) n (m + 1)
211
212
213  toallcocoons :: [ CState] -> Message -> ([Queue Message], [CState])
214  toallcocoons fs m = toallcocoons1 fs m (length fs)
215
216  toallcocoons1 :: [ CState] -> Message -> Int -> ([Queue Message], [CState])
217  toallcocoons1 fs m n | n == 0 = ([],[])
218                       | n > 0 = ((fst toall)++[(fst toone)],
219                                  (snd toall)++[(snd toone)])
220                         where toall = toallcocoons1 fs m (n - 1)
221                               toone = toonecocoon (fs!!(n - 1)) m n
222
223  toonecocoon :: CState -> Message -> Int -> (Queue Message, CState)
224  toonecocoon (cs, rb, pb, fid) m n = cocoon cs rb fid n m pb
225
226  -- operates purely on tree
227  prune :: Augtree Message -> DFA Message -> Augtree Message
228  prune t rules = pruneTree rules t
229
230  -- operates purely on tree
231  extractdup :: Augtree Message -> Augtree Message
232  extractdup t = extractdupTree t
233
234  -- operates purely on tree
235  extract :: Augtree Message -> Augtree Message
236  extract t = extractTree t
237
238  commit :: Augtree Message -> [CState] -> ([Message], [CState])
239  commit m fs = (commituser (traverseAugtree m),
```

```
240                  snd(toallcocoons (setfeatures (traverseAugtree m) fs) (Rcv, Commit_transaction, Nil, 0)))
241
242  commituser :: [( Message, Set Int)] -> [Message]
243  commituser [] = []
244  commituser (m:ms) = fst(m):(commituser ms)
245
246  setfeatures :: [( Message, Set Int)] -> [CState] -> [CState]
247  setfeatures m fs = sf1 (fst (split m)) (set2List (snd (split m))) fs
248
249  sf1 :: [ Message] -> [Int] -> [CState] -> [CState]
250  sf1 ms active fs = sf2 ms active fs (length fs)
251
252  sf2 :: [ Message] -> [Int] -> [CState] -> Int -> [CState]
253  sf2 ms active fs n | n == 0 = [] -- we have dealt with all features
254                     | n > 0 && ismem = setfs ++ [setactivefeat ms (fs!!(n - 1)) n]
255                     | n > 0 && not (ismem) = setfs ++ [setinactivefeat (fs!!(n - 1)) n]
256                       where ismem = isMemberSet (list2Set active) n
257                             setfs = (sf2 ms active fs (n - 1))
258
259  --put features in right state, i.e. start commit and replay msgs (setactive1)
260  setactivefeat :: [ Message] -> CState -> Int -> CState
261  setactivefeat m (cs, rb, pb, fid) n = setactive1 m (snd(cocoon cs rb fid n (Rcv, Commit_transaction, Nil, n) pb)) n
262
263  setactive1 :: [ Message] -> CState -> Int -> CState
264  setactive1 [] (cs, rb, pb, fid) n = (cs, rb, pb, fid)
265  setactive1 [x] (cs, rb, pb, fid) n = snd(cocoon cs rb fid n x pb)
266  setactive1 (x:xs) (cs, rb, pb, fid) n = setactive1 xs (snd(cocoon cs rb fid n x pb)) n
267
268  --abort past actions
269  setinactivefeat :: CState -> Int -> CState
270  setinactivefeat (cs, rb, pb, fid) n = snd(cocoon cs rb fid n (Rcv, Commit_transaction, Nil, n) pb)
271
272  split :: [( Message, Set Int)] -> ([Message], Set Int)
273  split [] = ([], emptySet)
274  split [(x, n)] = ([x], n)
275  split ((x, n):xs) = (x : fst (split xs), unionSet (snd (split xs)) n)
```

Bibliography

[AA97] P. K. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In *[DBL97]*, pages 153–167, June 1997.

[ABB+93] M. Arango, L. Bahler, P. Bates, M. Cochinwala, D. Cohrs, R. Fish, G. Gopal, N. Griffeth, G. E. Herman, T. Hickey, K. C. Lee, W. E. Leland, C. Lowery, V. Mak, J. Patterson, L. Ruston, M. Segal, R. C. Sekar, M. P. Vecchi, A. Weinrib, and S. Y. Wuu. The Touring Machine System. *Communications of the ACM*, 36(1):68–77, January 1993.

[AC97] I. Aggoun and P. Combes. Observers in the SCE and SEE to detect and resolve feature interactions. In *[DBL97]*, pages 198–212, June 1997.

[ACC+00] D. Amyot, L. Charfi, N. Corse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and lotos. In *[CM00]*, pages 274–289, May 2000.

[AGG+98] A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne. SCF3^{TM}/Sculptor with Chisel: Requirements engineering for communications services. In *[KB98]*, pages 45–63, September 1998.

[AKGM00] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In *[CM00]*, pages 94–112, May 2000.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, Reading (Massachussets), 1986.

[BAE+98] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Feature-interaction visualization and resolution in an agent environment. In *[KB98]*, pages 135–149, September 1998.

[Bag01] D. Bagley. The Great Computer Language Shootout, `http://www.bagley.org/ dough/shootout`, 2001.

[BB01] L. Blair and G. Blair. Specifying and analysing multimedia systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing.* Cambridge University Press, Cambridge, 2001. ISBN: 0521771846.

[BBPE01] L. Blair, G. Blair, J. Pang, and C. Efstratiou. 'feature' interactions outside a telecom domain. *Workshop on Feature Interactions in Composed Systems - held at ECOOP2001*, 2001.

[BDC+89] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y-J. Lin. The feature interaction problem in telecommnucation systems. *7th International Conference on Software Engineering for Telecommunications Systems*, pages 59–62, July 1989.

[BJK94] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In L. G. Bouma and H. Velthuijsen, editors, *[BV94]*, pages 197–216, May 1994.

[BP00] L. Blair and J. Pang. Feature interaction – life beyond traditional telephony. In *[CM00]*, pages 83–93, May 2000.

[BR01] L. Blair and S. Reiff-Marganiec. Runtime resolution of interaction of multimedia features. September 2001. Submitted for Publication.

[Bre00] J. Bredereke. Families of formal requirements in telephone switching. In *[CM00]*, pages 257–273, May 2000.

[BV94] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems.* IOS Press (Amsterdam), May 1994.

[CAD] CADP. Homepage for the CADP (Caesar/Aldebaran Development Package) toolkit , `http://www.inrialpes.fr/vasy/cadp.html`. Valid on 10-12-2001.

[Cai92] M. Cain. Managing run-time interactions between call processing features. In *IEEE Communications Magazine*, pages 44–50, February 1992.

[Cal98] M. Calder. What use are formal design and analysis methods to telecommunications services? In *[KB98]*, pages 23–31, September 1998.

[CGL+94] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for IN and beyond. In *[BV94]*, pages 1–23, May 1994.

[CKMR01] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, October 2001. Submitted for Publication.

[CM00] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI.* IOS Press (Amsterdam), May 2000.

[CM01] M Calder and A Miller. Using SPIN for feature interaction analysis - a case study. In *[Dwy01]*, pages 143–162, 2001.

[CMM99] M. Calder, E. Magill, and D. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *IEE Proceedings - Software*, 146(3):167–180, June 1999.

[CMRT02] M. Calder, E. Magill, S. Reiff-Marganiec, and V. Thayananthan. Theory and practice of enhancing a legacy software system. In Peter Henderson, editor, *Systems Engineering Business For Process Change - New Directions*, pages 120–137. Springer Verlag, London, 2002.

[CO95] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications Systems III.* IOS Press (Amsterdam), October 1995.

[CR00] M. Calder and S. Reiff. Modelling legacy telecommunications switching systems for interaction analysis. In Peter Henderson, editor, *Systems Engineering Business Process Change*, pages 182–195. Springer Verlag, London, May 2000.

[DBL97] P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV.* IOS Press (Amsterdam), June 1997.

[dORZ98] L. de Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation: a synchronous point of view. In *[KB98]*, pages 262–275, September 1998.

[Dwy01] M.B. Dwyer, editor. *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, Toronto, Canada, May 2001. Springer Verlag.

[FN00] A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In *[CM00]*, pages 179–192, May 2000.

[Fri95] N. Fritsche. Runtime resolution of feature interactions in architectures with seperated call and feature control. In *[CO95]*, pages 43–63, October 1995.

[GBGO98] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta. Feature interaction detection contest. In *[KB98]*, pages 327–359, September 1998.

[ghc] ghc. The Glasgow Haskell Compiler homepage, `http://www.haskell.org/ghc`. Valid on 10-12-2001.

[Gib97] J. P. Gibson. Feature requirements models: Understanding interactions. In *[DBL97]*, pages 46–60, June 1997.

[GV94] N. D. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *[BV94]*, pages 217–236, May 1994.

[Hal98] R. J. Hall. Feature combination and interaction detection via foreground/ background models. In *[KB98]*, pages 232–246, September 1998.

[Hal00] R. Hall. Feature interactions in electronic mail. In *[CM00]*, pages 67–82, May 2000.

[Has] Haskell. Homepage for the Haskell functional programming language `http://www.haskell.org`. Valid on 10-12-2001.

[HS88] S. Homayoon and H. Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. *IEEE Communications Magazine*, pages 42–46,70, December 1988.

[HS98] M. Heisel and J. Souquières. A heuristic approach to detect feature interactions in requirements. In *[KB98]*, pages 165–171, September 1998.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol. *Request for Comments (Proposed Standard) 2543*, 1999.

[Hug] Hugs. Haskell Users Gofer System homepage. `http://www.haskell.org/hugs`. Valid on 10-12-2001.

[ISO89] ISO. *ISO standard 8807:* LOTOS *– A formal description technique based on the temporal ordering of observational behaviour.* ISO/IEC, 1989.

[ITU92] International Telecommunications Union ITU. *ITU-T Recommendation Q.1201: Principle of Intelligent Network Architecture.* ITU-T, October 1992.

[ITU93a] International Telecommunications Union ITU. *ITU-T Recommendation Q.1204: Intelligent Network Distributed Functional Plane Architecture.* ITU-T, 1993.

[ITU93b] International Telecommunications Union ITU. *ITU-T Recommendation Q.1214: Distributed Functional Plane for Intelligent Network CS-1.* ITU-T, March 1993.

[ITU97] International Telecommunications Union ITU. *ITU-T Recommendation Q.1221: Introduction to Intelligent Network Capability Set 2.* ITU-T, September 1997.

[ITU00] International Telecommunications Union ITU. *ITU-T Recommendation H.323: Packet Based Multimedia Communications Systems (Version 4).* ITU-T, November 2000.

[JAI] JAIN. Java API's for Integrated Networks homepage `http://java.sun.com/products/jain`. Valid on 10-12-2001.

[KB98] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V.* IOS Press (Amsterdam), September 1998.

[KB00] A. Khoumsi and R. Bevelo. A detection method developed after a thorough study of the contest held in 1998. In *[CM00]*, pages 226–240, May 2000.

[KCK+95] B. Kelly, M. Crowther, J. King, R. Masson, and J. DeLapeyre. Service validation and testing. In *[CO95]*, pages 173–184, October 1995.

[Kim97] K. Kimbler. Addressing the interaction problem at the enterprise level. In *[DBL97]*, pages 13–22, June 1997.

[Kim00] K. Kimbler. Service interaction in next generation networks: Challanges and opportunities. In *[CM00]*, pages 14–20, May 2000.

[KK98] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[KK00] M. Kolberg and K. Kimbler. Service interaction management for distributed services in a deregulated market environment. In *[CM00]*, pages 23–37, May 2000.

[KKM94] K. Kimbler, E. Kuisch, and J. Muller. Feature interactions among pan-european services. In *[BV94]*, pages 73–85, May 1994.

[KKV64] W. Keister, R. W. Ketchledge, and H. E. Vaughan. No. 1 ESS: System organisation and objectives. *Bell Systems Technical Journal*, 43(5):1831–1844, 1964.

[KL98] J. Kamoun and L. Logrippo. Goal-oriented feature interaction detection in the intelligent network model. In *[KB98]*, pages 172–186, September 1998.

[KMMR00] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Second feature interaction contest. In *[CM00]*, pages 293–310, May 2000.

[KS94] K. Kimbler and D. Sobirk. Use case driven analysis of feature interactions. In *[BV94]*, pages 167–177, May 1994.

[LOL] LOLA. (LOtos LAboratory), `http://yeti.dit.upm.es/ lotos/tools/lola.html`. Valid on 10-12-2001.

[LS00] J. Lennox and H. Schulzrinne. Feature interaction in internet telephony. In *[CM00]*, pages 38–50, May 2000.

[Luc] Lucent Technologies – Bell Labs Innovations. Pathstar access server. `http://www.lucent.com/ins/products/pas`. Valid on 30-09-2001.

[Mar00] D. Marples. *Detection and Resolution of Feature Interactions in Telecommunications Systems at Runtime.* PhD Thesis, Communications Division, Department of Electrical and Electronic Engineering, University of Strathclyde, 2000.

[MM98] D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *[KB98]*, pages 115–134, September 1998.

[MMS95] D. Marples, E. H. Magill, and D. G. Smith. An infrastructure for feature interaction resolution in a multiple service environment - the application of transaction processing techniques to the feature interaction problem. In *Proceedings of TINA 95 conference*, 1995.

[MTMS95] D. Marples, S. Tsang, E. H. Magill, and D. G. Smith. A platform for modelling feature interaction detection and resolution techniques. In *[CO95]*, pages 185–199, October 1995.

[MY60] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1):39–49, 1960.

[Mye88] E. W. Myers. A four-russians algorithm for regular expression pattern matching. *University of Arizona Technical Report*, TR 88–34, 1988.

[NKHL00] M. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo. Feature interaction filtering with use case maps at requirements stage. In *[CM00]*, pages 163–178, May 2000.

[Par] Parlay. The Parlay API, `http://www.parlay.org`. Valid on 10-12-2001.

[PR98] M. Plath and M. Ryan. Plug-and-play features. In *[KB98]*, pages 150–164, September 1998.

[Pre97] C. Prehofer. An object-oriented approach to feature interaction. In *[DBL97]*, pages 313–325, June 1997.

[Pyt] Python. Python programming language homepage. `http://www.python.org`. Valid on 10-12-2001.

[Rei00] S. Reiff. Identifying resolution choices for an online feature manager. In *[CM00]*, pages 113–128, May 2000.

[Rei01] S. Reiff-Marganiec. Using LOTOS for modelling and analysing an online feature manager: A study. June 2001. Department of Computer Science, University of Glasgow, Glasgow(UK), TR-2001-92.

[RH97] S. M. Rochefort and H. J. Hoover. An exercise in using constructive proof systems to address feature interactions in telephony. In *[DBL97]*, pages 329–341, June 1997.

[RV94] F. J. Redmill and A. R. Valdar. *SPC: Digital Telephone Exchanges.* Peter Peregrinus Ltd. (Stevenage), 1994.

[SIG01] Bluetooth SIG. Specification of the bluetooth system: Version 1.1. February 2001.

[SL95] B. Stepien and L. Logrippo. Representing and verifying intentions in telephony features using abstract data types. In *[CO95]*, pages 141–155, October 1995.

[Tho97] M. Thomas. Modelling and analysing user views of telecommunications services. In *[DBL97]*, pages 168–182, June 1997.

[Tho99] S. Thompson. *Haskell – The Craft of Functional Programming.* Addison Wesley, 2nd edition, 1999.

[Tho01] S. Thompson. Regular expressions and automata using Haskell. *University of Kent Technical Report*, TR 5–00, 2001.

[TM97] S. Tsang and E. H. Magill. Behaviour based run-time feature interaction detection and resolution approaches for intelligent networks. In *[DBL97]*, pages 254–270, June 1997.

[TM00] V. Thayananthan and E. Magill. A practical filtering technique for triple feature interactions. 2000.

[TMK97] S. Tsang, E. H. Magill, and B. Kelly. The feature interaction problem in networked multimedia services - present and future. *BT Technology Journal*, 15(1):235–246, January 1997.

[Tur98] K. J. Turner. Validating architectural feature descriptions using LOTOS. In *[KB98]*, pages 247–261, September 1998.

[Tur00] K. Turner. Formalising the Chisel notation. In *[CM00]*, pages 241–256, May 2000.

[Vel93] H. Velthuijsen. Distributed artificial intelligence for runtime feature interaction resolution. *Computer*, 26(8):48–55, August 1993.

[VGL92] H. Velthuijsen, N. Griffeth, and Y.-J. Lin, editors. *International Workshop on Feature Interactions in Telecommunications Software Systems*, December 1992. Not in Print.

[Wei89] W. E. Weil. Transaction processing techniques. In S. Mullender, editor, *Distributed Systems*, pages 329–352. ACM Press, New York, 1989.

[YO98] T. Yoneda and T. Ohta. A formal approach for definition and detection of feature interactions. In *[KB98]*, pages 202–216, September 1998.

[ZJ00] P. Zave and M. Jackson. New feature interactions in mobile and multimedia telecommunication services. In *[CM00]*, pages 51–66, May 2000.