IT Licentiate theses
2000-006

A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems

GUSTAF NAESER

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems

BY
GUSTAF NAESER

September 2000

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2000

A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems

Gustaf Naeser
gaffe@docs.uu.se

Department of Computer Systems
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

The complexity of today's telecommunications systems grows with each new feature introduced. As the number of services an operator provides can be used to gain advantage over competitors the number of services will continue to increase. As the complexity grows, so does the possibility for feature interactions, situations where the operation of features interfere with the operation of other features. Interactions cost more to resolve the later during the introduction of a new feature they are detected. This makes the need for analysis of feature interactions during development preeminent.

There exists a multitude of frameworks and techniques for specification and analysis of models of telecommunications systems. However, we have identified that they often impose unnecessary design decisions on the models, making them untoward.

In this thesis we propose a framework for specification of models of telecommunications systems. The framework is designed to describe them as general systems of communicating processes in a flexible way which allows alterations to be made late during the design. We also present definitions of interactions and how the interaction definitions are used in the framework to detect undesired interactions.

A model for telephony systems is derived through observations made of common telephony concepts and constructs. Delving into concepts early in the design of the system is shown to avoid several sources of interactions.

To demonstrate the potential of the framework we present a case study where the system and services described in the first interaction detection contest has been implemented and searched for interactions.

Acknowledgments

First I would like to thank my supervisor Prof. Bengt Jonsson for accepting me as a Ph.D. student and introducing me to the world of formal methods. Special thanks goes for him being a source of inspiration and to all his invaluable comments on my writings, especially this thesis. His infectious enthusiasm is a great motivation.

A thank goes to Jan Nyström for being the often much needed "bollplank" on which great ideas can be thrown to disappear. Much of the implementation work in this thesis has been done with him and like Pooh says, "It's more fun being two".

Thanks goes to all past and present colleges at DoCS who are a constant reminder of the greatness of academia. Special thanks goes to Kristina Lundqvist, Göran Wall, Björn Knutsson and Mikael Sjödin for all dinners and gaming sessions we've had; the best way to forget the worldly stuff is often to satiate your stomach and then plunger into eight solid hours of friendly company by a silver screen.

Last but not least I am grateful for the support and faith my parents and siblings always have given and shown me. There would be no meaning without you.

Publications

This thesis is based on work, parts of which have been previously published. The following list contains the relevant papers and notes on my participation in them.

[A] G. Naeser , J. Nyström, B. Jonsson, "A Case Study in Automated Detection of Service Interaction", *Proceeding of RadioVetenskap och Kommunikation 99*, 1999.

[B] G. Naeser, J. Nyström, B. Jonsson, "Building Tools for Creation and Analysis of Telephone Services", *Proceeding of RadioVetenskap och Kommunikation 99*, 1999.

[C] B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen, "On Modelling Feature Interactions in Telecommunications", *Proc. Nordic Workshop on Programming Theory '99)*, B. Victor and W. Yi, eds., 2000.

[D] B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen, "Incremental Requirement Specification for Evolving Systems", *Feature Interactions in Telecommunications and Software Systems VI*, M. Calder and E. Magill, eds., pp. 145–162, ISO Press, 2000.

Paper [A] is an early description of the model and my participation in the paper was major design, joint implementation with Jan Nyström, and most writing.

Paper [B] is about paper building blocks for the ITU-T IN model of telecommunications systems. My participation to this paper has been implementation and advisory.

Paper [C] is in a collection of extended abstracts which are to be published in full. The paper is about using requirements to detect interactions. My contribution was implementation, testing and major parts of design and writing.

Paper [D] is a paper related to Paper [C] focusing on how requirements are iteratively tuned to the system. My participation in this paper has been the same as for Paper [C].

Contents

List of Figures

List of Tables

Symbols

c a control location.

e an event type.

e an event.

$\mathcal{E}$ a set of events, most often the response to an event.

g a guard.

$\_$ an anonymous variable (used when the value of a variable is not needed).

p a process configuration.

σ a mapping from the variables to values.

$\overline{d}$ a tuple $d_1, \ldots, d_i$ of parameters.

$\overline{u}$ a tuple $u_1, \ldots, u_i$ of variables.

$\oslash$ the priority operator which removes events of lower priority.

$prio(\mathcal{E})$ the priority events in $\mathcal{E}$.

$p \xrightarrow{e/\mathcal{E}} p'$ a process step triggered by the event e.

Chapter 1

Introduction

In this thesis, I present a framework for detection of feature interaction.

Feature interaction occurs in a telecommunications when one feature modifies or subverts the operation of other features. This kind of behaviour is not limited to the world of telecommunication; feature interaction can occur in any system where there are agents with different goals.

The problem with feature interaction can be explained to a human in a matter of minutes, explaining it to a computer is not so easy. However, whereas a human would tire after looking for interactions in a system for an extended amount of time a computer would not and would carry on searching until finished or told otherwise. In this thesis, telephony systems will be the system in which feature interactions are searched for. Telephony systems can be complex and it can be hard to envision all the interactions that can occur between the different telephony system components. Using a good model of the system computers can exhaustively search for feature interactions. Telecommunication systems are an interesting field for study of feature interactions thanks to the large number of features, interactions and ease for humans to reason about the interactions.

The growing competition between operators in the telecommunication market forces them to look for ways to excel. One way to get an edge is to offer more and better features to the customers. But each introduction of a feature into a system can also introduce undesired interactions. To avoid compromising the quality of the service provided, the operators need to ensure that no unwanted interactions are introduced along with the new features. This makes the problem of detecting (and resolving) feature interaction important.

A lot of attention has been devoted to the development of methods for detecting and resolving feature interactions in recent years. The complexity of the systems, and the fact that they grow in complexity with each new feature introduced, implies that a good framework is needed.

My research has been focused on the simple question: *How can the introduction of new telecommunication features into an existing feature set be accomplished without*

introducing undesired interactions? Asking this question suggests that it should be detected whether the introduction of a new feature will also introduce interactions before the feature is actually introduced into the operating system. The detected undesired interactions must also be resolved.

There are three categories of approaches, described by Cameron and Velthuijsen [CaVe93], to handling the interaction problem: detection, resolution, and avoidance. Detection aims at finding and reporting interactions present in a system, resolution introduces means to solve interactions that have been identified, and avoidance (also known as prevention) avoids situations where interactions occur. From this we understand that there is no need for resolution in a system with avoidance.

This thesis is concerned with detection and resolution.

In any solution to the interaction problem, there are common tasks that need to be addressed. The notion of "what is an interaction" has to be defined, the system in which interactions are to be detected (here a telephony system) needs to be described, and the method for detecting and resolving interactions need to be described.

The term *interaction* is normally used to denote *undesired interactions* only. Introducing a feature into a system will change the behaviour of the original system and thus the new feature will cause an interaction with the system. But the interaction caused by the activation of the new feature was an intended when the feature was added, the interaction was *desired* and thus this is a harmless interaction. What really needs to be detected is whether the introduction of the new feature raises unexpected interactions that for instance will prevent features already present in the system to run as intended.

The research community has put effort in to both describing feature environments and creating methods for checking that systems are correct with respect to given criteria and of course different approaches to describing the system will of course influence how interactions are detected.

A framework for detecting feature interactions should be general and able to describe many different systems where feature interactions can occur, it should not be restricted to telecommunication systems. This requires that the framework is based on generic techniques for defining features and for analysing communicating systems.

The contributions presented in this thesis are:

- a flexible framework for describing telecommunication systems and features as communicating systems,
- three ways of detecting feature interactions in the framework, and

- examples of how the framework and the interaction definitions can be applied to large communication systems.

The goal of the framework has been to create an environment in which a communicating system can be easily described, modified and analysed. The framework allows the models level of granularity to be increased (or decreased) at need allowing new concepts are easy to introduced. Avoiding interactions introduces solely by the framework has also been imperative. The order in which features are added to the model can, in other frameworks, also introduce priority and subversion of features already present, and a goal has been to avoid this. The framework is described in Chapter 4.

The definitions of interactions as well as the rationale of them are presented in Chapter 5.

Application of the framework is presented in Chapter 6 using a case study of the first feature interaction detection contest [GBG+98]. A model of a telecommunication base system and a set of features are given and the results from the interaction detection is discussed. The model is one of several possible—one we found it comfortable working with.

However, the thesis starts with an introduction of the topic using three example scenarios in which interactions are illustrated, Chapter 2. Each scenario will give an example of a different aspect of feature interactions and taken together the examples will serve as a base for the understanding of the more technical parts of the later discussion. The chapter will introduce a model of telecommunication systems which is constructed by observing and reasoning about the components, constructs and concepts found in a generalised telecommunication system.

The description of the approach will refer to the examples whenever they can be used to clarify a construct or design decision.

Chapter 2

Three Examples

Someone who has never thought about feature interactions can in minutes understand the outline of a telecommunication feature, one type of interaction and how that interaction can occur in a scenario with the feature. Explaining the same thing to a computer is not as easy. Though the capability of the human minds reasoning cannot be matched by computers, computers can search large amounts of data faster than humans. Deciding whether or not a collection of features contains interactions is a very large search so much can be gained by using computers.

This chapter introduces the area using three examples of interactions. The examples, *Example One*, *Example Two* and *Example Three*, will then be used as a basis for a discussion of different concepts in the area of telecommunication systems and feature interactions. The purpose of this chapter is to give an introduction to features, terminology and interactions. Deeper discussions of the topics are given in later chapters.

2.1 EXAMPLE 1: THE FORWARDING LOOP

Fredrik is a computer scientist who lives in the city where he works. On weekends he often goes to his summerhouse that is located just some hours away. Since he works in maintenance, he must always be reachable and to make sure he is, he *forwards* incoming calls to the place where he is.

The IN Call Forwarding[1] service he uses is a simple one that just forwards incoming calls to a given number.

Peter wants to call his good friend Fredrik. He picks up the receiver and dials Fredrik's number. Fredrik is not at home, he is sitting in his car on the way to his summerhouse and he has forwarded his phone to the summerhouse. Normally this would not be a problem. However, Fredrik has forwarded the phone in his summerhouse to his phone in the city apartment and has not yet reached that phone to cancel that forwarding.

[1] IN stands for Intelligent Network which we for now can take to mean *a telephony system with features in it.*

Peter's call is first forwarded from the apartment to the summerhouse, then back again to the apartment, to the summerhouse, back again, to the summerhouse, back again...

2.2 EXAMPLE 2: THE AMBIGUOUS HASH

Malin is abroad. Since dialling from other countries often implies large of amounts change and constantly feeding them to the coin slot, she has started subscribing to an Account Card Calling service.

The Account Card Calling service Malin subscribes to works in the following way:

1. The subscriber calls the predefined free number of the operators service.
2. The service starts an authorisation procedure where it asks for the account to be charged and a PIN code for the account. If no valid PIN is given the service will inform the caller that authorisation has failed and then the service will end the call.
3. When the user has completed the authorisation phase she will be rewarded with a dial tone and the ability to place calls.
4. During the call the user has the option to press the `hash` key, `#`, to start a new call. This extra feature has been added since the authorisation procedure, entering the account and PIN, can be time consuming and something you hate to do more than once every session.

Now, back to Malin.

Malin starts the service, completes the authorisation procedure without incidents, and then calls her own answering machine to see if anyone has left any messages on it. The answering machine, which supports remote control, demands that the owner identifies herself by entering a PIN before it allows remote playback and control. To tell the answering machine that all numbers of the PIN has been given a `hash` is entered. The `hash` is recognised by the answering machine, which then checks if the PIN is valid.

When Malin hits the `hash` after giving the answering machines PIN she will not be listening through messages as she expects. To her surprise, she gets a new dial tone and the connection to the answering machine is lost.

2.3 EXAMPLE 3: THE EXTRA BILLING

Tova has a cellular phone. Tova's operator has both a cellular network and a stationary one. To handle the different tariffs between cellular and stationary

calls, the operator has a base tariff for all calls and then uses a service called Cellular Phone to add an additional *air-time* tariff to cellular calls.

One day a fire starts in Tova's house. Tova uses her cellular phone to call the fire brigade. The number she dials is 112. The 112 service does many things of which the most important is connecting her to an available operator which can dispatch the closest fire brigade to save her home.

One feature of the 112 service is that it is free to call. This means that the callee, in this example the operator service, takes over the cost of the call. However, this is not known by the cellular phone service which still charges air-time to Tova while she is speaking to the 112 operator.

2.4 A BASIC VOCABULARY FOR FEATURES

We have now seen three scenarios which all have the common element that we think that something went wrong in them. Our first goal will be to establish a language that can be used to describe and discuss telecommunication systems. We will reason about the telecommunication systems and services described in the examples and from this discussion derive a vocabulary. Even if the examples seem innocent, they use terminology that must be defined if it is to be used in a language about telecommunication systems. Several terms have been introduced and in order to have a language that avoids misinterpretation, these and other terms should be given a specific meaning. The vocabulary started here will be further extended and specialised in later chapters. Zave [Zav93] points out that *telecommunications terminology is not standardised* and that this often leads to confusion when discussing the area.

The aim of the vocabulary is to define concepts we need in order to translate the real world problem to a model of the systems which can be understood and analysed by a computer. The vocabulary will also help in preventing service specifications from being misinterpreted. Some terms are introduced to help reasoning about the telecommunication systems whereas others will be directly used in the model.

Our model will be based on the concepts identified and defined in the vocabulary and since there are numerous interpretations of the concepts, the model will be one in the multitude of possible ones. The rationale for the model we use is it in easy and natural ways captures and models the simple concepts of telecommunications. Some of the aspects of the model will not be specific to telecommunication systems and can therefore be used in a wide range of systems that share the property captures by the modelled concept.

We start by defining the difference between a *feature* and a *service* which we will use. This separation of concepts is close to the one made by Cameron and Velthuijsen [CaVe93].

Feature A feature is functionality which extends the base system (i.e. the telephony system). Features can be seen as building blocks; by themselves features does not implement functionality that can be sold to subscribers but they can be combined to implement more sophisticated functionality.

Service A service is a collection of features packaged and interconnected to implement a specific behaviour. Services constitute functionality that can be offered to subscribers. You do not subscribe to features, you subscribe to services.

A sample feature could be Give PIN which implements the functionality to receive a `PIN` from the user. Alone, this feature can not be sold to a subscriber, but combined with other features it can be used to build services like the Account Card Calling service.

In *Example One*, where Peter's connection attempt is being ping-ponged between Fredrik's apartment and summerhouse, there is only one service, IN Call Forwarding. As we will later see, this service actually consists of a feature for Call Forwarding and some functionality for activation of the service. The forwarding feature can be used with other triggering conditions, i.e. conditions for the service to be activated, to build other forwarding services. The triggering condition in the example is unconditional but an easy condition could be that forwarding only should occur when the called phone is busy, like in the service Call Forwarding on Busy.

Example Two contains two services, Account Card Calling and an answering machine. We can treat the answering machine as if it were a service in the system since the only difference would be that Malin would not have yet another piece of electronic equipment at home.

The service in the third scenario is 112.

All features we define and use are ones we have decided to have in our model and are not necessarily the ones found in real world systems or even in other models. The features we use have been designed and are used since we find them useful. A good thing with features is that some of them can be reused, something often desired since it can reduce development time.

The term service will be used sparsely and only when discussing the behaviour implemented by a composition of features. Most often the term feature will do.

2.4.1 Communicating Systems

Like many other systems, a telecommunication system can be described as a set of components that interact to implement telephony. Components inform other components of changes in the system. A sample scenario could be that

1. A phone tells the switch that its receiver has been lifted.
2. The switch confirms by starting a dial tone in the phone.
3. The phone wants to start a call and sends a request to the switch.
4. The switch tells the called phone to start ringing its bell and also tells the calling phone to start a ringing tone.
5. The called phone accepts the call and notifies the switch of this.
6. The switch creates a connection between the two phones, and notifies them about the connection. It also tells the billing system to start billing for a call between the caller and the callee.

We can identify three different hardware components and a set of rules for how they should react to different stimuli in the example above. The example is illustrated in Figure 2.1 where the arrows represent the messages and the numbers on the arrows are when the messages are sent according to the sample scenario above.

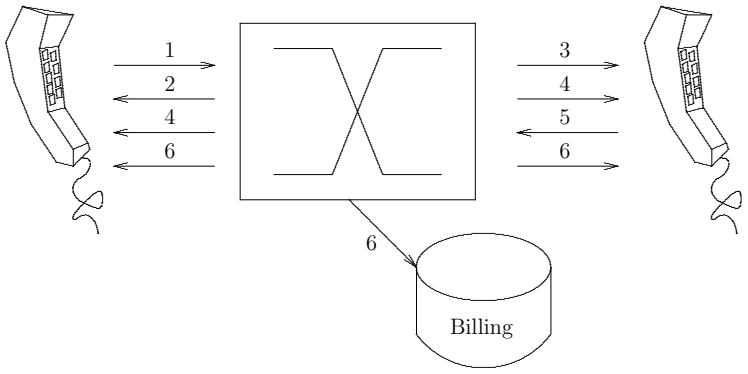

Figure 2.1: A small communication example.

The hardware components are a set of phones, the switch and the billing system. It is possible to extend the set of components with specialised databases or other miscellaneous resources, but for now we will only use the three components listed above. The set of rules used to decide how the components should react are the features. The feature described above is a plain old telephony call.

More specifically a telephony system is a reactive system [MaPn95]. A reactive system is a system where components react to stimuli presented to them and in response they may generate stimuli to other components.

Describing a telecommunication system in a way that can be implemented in a tool and run on a computer will include modelling both the hardware and the software components of the system.

We will identify two important properties separate hardware and software components. Designing the framework to support both, as opposed to supporting only on of them and coding the other in that framework, can make modelling easier.

Persistence The hardware components of the base system are persistent[2].

Change Changing the behaviour of functionality of a hardware component can have severe impact on software components. If functionality is removed from or added to a hardware component excessive changes may be needed in other components.

In this discussion hardware components will exist during a whole scenario. If there is a phone there will always be a phone. Software components, on the other hand, may be created and removed at will. This makes the modelling flexible but it also makes the modelling more complex.

Change is an important issue since components depending on the implementation of base components may have to be changed when the base components change. Imagine the modifications needed if the billing system is redesigned to accept *terminate billing* messages instead of *end billing* messages or the changes needed if the way connections are set up is changed. In the first case the old message would have to replaced everywhere by the new message, a task that can be hard if the message is used in a great number of places, and in the second case features may have to be completely rebuilt to be compatible with the new connection model.

To help overview the changes of a system we impose a layered view of our model of the system. Components depending on other components are placed in higher layers. The lowest layer contains the most basic functionality needed by all other components. In our model of telecommunication system features reside in layers higher than the basic switching and connection handling since change in, e.g., connection handling may require that features are modified to use the changed connections.

[2]Other components may also be persistent, but *all* hardware components are persistent.

2.4.2 The Base System

The term *base system* will be used to denote a simple telephony system consisting of three types of components: a switch, a billing system and a set of phones. The base system constitutes the lowest layer of our model.

Switch The switch

- passes events between system components, both base system components and features,
- does basic bookkeeping like keeping track of current connections, i.e. it knows if there exists a connection between two phones,
- keeps track of subscriptions (like which subscriber subscribes to what service and with what parameters), and
- runs features.

Another functionality we have in the switch is the functionality for relaying recorded announcements, like ‘‘`Give PIN`’’. In our model there is no other obvious place for this functionality and it seems intuitive to house it in the switch which handles connections.

Our framework allows the switch to be extended with additional functionality when needed, but since it is part of the lowest layer it is desirable to limit its complexity since it will have impact on higher layers. For example, the switch could have been responsible for keeping track of billing, but since billing has a clear interface, implementing the billing in a stand alone component makes good sense and was the design we chose.

The *events* passed between the system components are simple messages used to inform the other component that something had or should occur. A sample event is the *end billing* event (which in our model is written $\not\downarrow billing$). Events can have parameters to indicate, e.g., a specific phone affected by the event.

Billing system The billing system is a database which records billing in the system. Features can start and stop billing by generating specific stimuli, events, directed to the billing system.

The functionality provided by the billing system can be extended when needed. An example of such functionality is split billing. The new functionality has to be introduced since it embodies functionality that can not be implemented without it.

Phones Phones are models of how users are allowed to operate the switch. They have a receiver, number buttons and special buttons like `hash` an `flash`. Phones can be used to

- dial numbers
- emit tones and voice through the receiver
- sound bell signals

The `flash` button is used to generate a special signal used, e.g., when switching between two calls in the Call Waiting service. Phones have two ways of signalling. When the receiver is on hook they can activate the bell causing the phone to "ring" or when the receiver is off the hook they can emit tones, relay voice or playback messages to the user. Sample tones are the `dial` tone and the `busy` tone.

The phones are used to generate requests[3] to the switch and the task of the switch is to serve these requests. A sample request is to get a free line (recognised as a dial tone in the receiver). Section 2.4.4 will explain this in more detail.

Alone the base system cannot do anything. Nothing responds to the events generated by the phones and nothing generates events to the billing system. This is what we use features for. This can be compared to a computer (hardware) without software. With the correct software the computer can be used to do many things but without it the computer will just be a bunch of hardware components soldered together unable to do anything. In the same way, the base system can do many things depending on the software (features) it is running.

Though the example scenarios do not specifically mention base system components, we know that they need to be there; the switch is needed for the forwarding and the billing system for all charging including the added air-time tariff in the third scenario.

Khoumsi [Kho97] stresses the importance of having a separate description of the base system since it remains the same regardless of the services currently running.

2.4.3 Connections

The switch we introduced has the ability to run features and the first service we want to introduce will be one which implements the a basic call model which allows users to other users without any bells or whistles. In order to be able to speak about functionality of the call model we need to define what a *connection* is.

We earlier stated that changes to a base system concept may require that components depending on the changed concept are modified. This prompts us to directly introduce *connection legs*.

Connection Leg A connection leg is a direct connection between two features.

[3] Another describing name for this is *external phenomena*[Zav98].

Connection A connection is an acyclic path consisting of one or more connection legs between two features.

The components connected by the connection legs are features that can relay speech to a phone if they want to.

Until now we have used *caller* and *callee* to denote the phones at the ends of the connection, but these terms will not be enough when working with connection legs.

Caller The caller is the phone, or feature, that initiated the connection. The caller is the same for all connection legs of a connection and the connection itself.

Callee The callee is the phone which accepted or rejected the call. The callee is the same for all connection legs of a connection and for the connection itself.

The connection legs themselves have a start and end, not necessarily the caller and callee, and we will need to be able to talk about these.

Originator The originator of a connection leg is phone which sent teh request. Note that a service can have sent or forwarded the request on behalf of the phone.

Terminator The terminator of a connection leg is where the request was sent, where connection leg ends.

Services are either on the originating side, acting as originators, or terminating side, acting as terminators.

In *Example One* the IN Call Forwarding service is operating at the terminating side. When a connection attempt reaches Fredrik's phone it reacts by starting the forwarding service.

The service in *Example Two*, Account Card Calling, is an originating side service. The service is called and then it enters a state where calls can be initiated.

A connection has a life consisting of three phases, the *connecting phase*, the *waiting phase* and the *connected phase*.

Connecting phase The connecting phase starts when a connection can be attempted and ends when it is known whether the connection will be established or not. A connection will be rejected, i.e. not established, if the

initiator cancels the attempt by hanging up, or if the terminating side decides to reject the call. If a connection can be established the terminating side responds that it can accept the connection and the connection enters the *waiting Phase*. A third possibility is that the terminator adds a connection leg and defers the decision of the terminating side to another terminator.

Waiting phase The waiting phase starts when the terminating side responds that it can accept the connection and ends either when the originating side aborts the connection or the terminating side establishes a connection. A normal way of establishing a connecting is by picking up the receiver to answer the incoming call. When the connection is established the connection enters the *connected Phase*.

Connected phase The connected phase starts when the terminating side accepts the incoming call and ends when either the terminating side or originating side ends the connection.

Features will be active during the different phases. The forwarding service in *Example One* is typically active during the connecting phase, the credit card service is active during the whole life of a connection as is the 112 service.

The example scenarios give two illustrations of when the connection phases are perverted. In *Example One* the connecting phase is never ended and in *Example Two* the connected phase is prematurely—and to Tova surprisingly—ended.

The specific events used in connections are introduced below when discussing them in our first service.

2.4.4 Plain Old Telephony System (POTS)

Knowing about connections, we observe how real world telephony systems work. We will describe and model a view of this behaviour in the most basic of all services whose name traditionally is Plain Old Telephony System, POTS. The service provides basic call functionality which allows users to pick up the receiver, dial numbers and get connections to the phones with which the numbers are associated.

It can be useful to separate originating side functionality from terminating side functionality and this will be done when describing POTS. We will later see that this will help reduce the complexity of the search for interactions.

The Originating Side of POTS (POTS_O)

Figure 2.2 shows how we designed the originating side of POTS, which will be called POTS_O, in our framework. Later, in Chapter 4, a more thorough definition of how

the graph works will be given, but for now we are satisfied with just understanding how it works in general. Figure 2.2 describes how POTS_O reacts to different

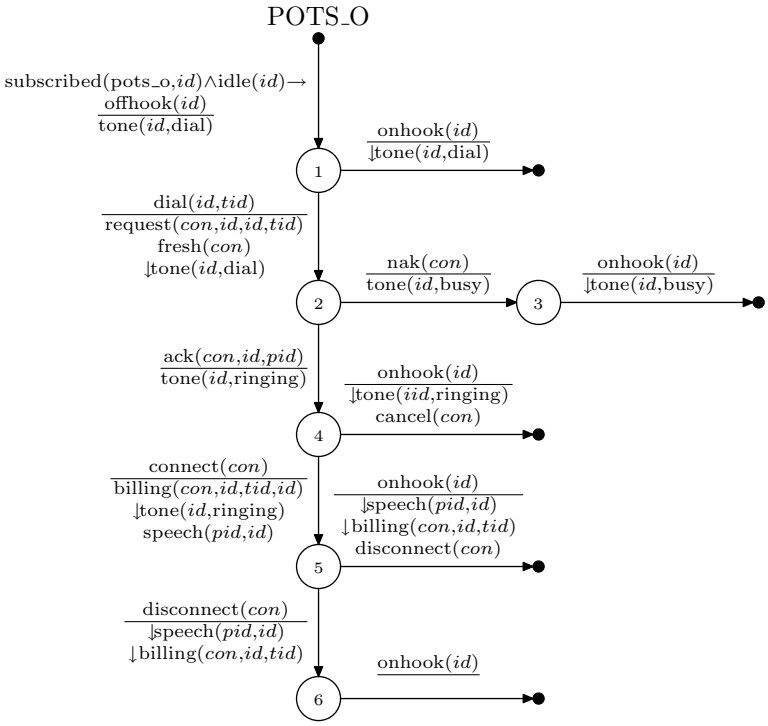

Figure 2.2: The originating side of POTS.

requests that can be presented to it during a connection. The control location with the label POTS_O describes the state of the originating side of a phone call when the originating phone is idle. We call this control location the *initial control location*. The arcs represent transitions to new control locations as response to stimuli.

An idle phone can generate a *request event* by lifting the receiver. The POTS_O feature generate a *response set of events* which starts a dial tone in the receiver. In Figure 2.2 this behaviour is described by the transition, arc, leading down from the initial control location ending in the control location labelled 1. The transition is also shown in Figure 2.3 and we will use this transition as an example transition

Request events Events are generated to request that someone takes action. For example, when the receiver is lifted on a phone an *offhook* event will be generated which will be the triggering event for transitions like the one shown in Figure 2.3 since it requests that action is taken to meet the change in the system.

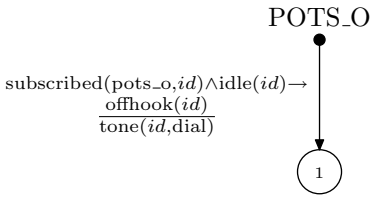

Figure 2.3: The first transition of POTS_O.

The *triggering condition of a transition* consists of two parts, a *guard* and a *triggering event*.

Guard The guard represents the condition for this particular arc to be activated and on the example transition it is $subscribed(pots\_o, id) \wedge idle(id)$. $subscribed(pots\_o, id)$ is a predicate to determine if the phone with *identity* id subscribes to the service and *idle* is a predicate used to determine if the phone is idle or not[4].

Triggering Event The triggering event on this transition is $offhook(id)$, indicating that phone id has lifted the hook. An arrow, $\rightarrow$, is used to separate the guard from the triggering event so that they are not confused. If the guard does not hold, in this case that the phone is not idle, the transition will not be used even if the triggering event is observed.

If a transition should be taken whenever the triggering event is observed it will have a guard which is always satisfied and the guard will not be explicitly written in such cases. All transitions, except for the triggering transition, of POTS_O are of this kind.

Guards can be used to check if, e.g., the correct PIN was submitted.

The *triggering transitions for a feature* are all the transitions that can start the feature. These transitions start in initial control locations, depicted with a dots labelled with the feature's name, and end in either a named control location or in a dot.

The *triggering conditions for a feature* consists of all the triggering conditions of the feature's triggering transitions. We say that a service is *armed* when a triggering condition for the feature is satisfied.

Response set of events The set of events generated in response to a request event is called the response set of events, or simply the response. This set

[4] $\wedge$ is a logic connective stating that both predicates have to be true for their conjunction to be true.

contains only one event, $tone(dial, id)$, on the example transition, a response which will request the phone to start emitting a dial tone.

The guard and request are separated from the response by a line just to make the separation of the triggering event and the responding events more visible.

We do not want several instances of POTS_O to be active for the same phone. Situations like that would make no sense. The $idle(id)$ predicate is used to ensure this: the predicate is true only if there is no POTS, originating or terminating, active for the phone with id id.

The *con* variable present in all events concerning connections is an identifier, a connection number that enables the switch to distinguish between different connections. The necessity of this identifier can be understood if we think about a case where there are two connection attempts to the same phone. Without the connection numbers the switch will not be able to tell them apart and will therefore not be able to reject one of them.

Looking at the originating side of POTS we can trace the phases of a simple connection and look at the events used in connection management, shown in Figure 2.4. The Figure shows the establishment of the connection *con* between the phones A and B. The Figure only shows the actions taken by the two phones and, e.g., not the actions by the billing system or the switch. Phone B uses the feature POTS_T which will be explained below.

Connection Phase The connecting phase starts when the *dial* event is observed (which takes POTS_O from control location 1 to control location 2). POTS_O will tell the rest of the system that it is entering this state by generating a special *request* event. The *request* is relayed to the terminating side so it can decide whether or not to accept the connection. The phase ends with either an *ack* or a *nak* event which tells the originator that a connection perhaps will be established respective that a connection is rejected. The control location will be 2 during the connecting phase.

Waiting Phase The waiting phase starts when the *ack* is observed and lasts until either the originator puts the receiver *onhook* or the terminating side accepts the connection by generating a *connect* event. If the originator hangs up a *cancel* event is generated to inform the terminating side of this. The control location will be 4 during the waiting phase.

Connected Phase The connected phase follows the waiting phase and is terminated by the *onhook* from the originator or by a *disconnect* from the terminating side, informing that the other party has hung up. The control location will be 5 during the connected phase.

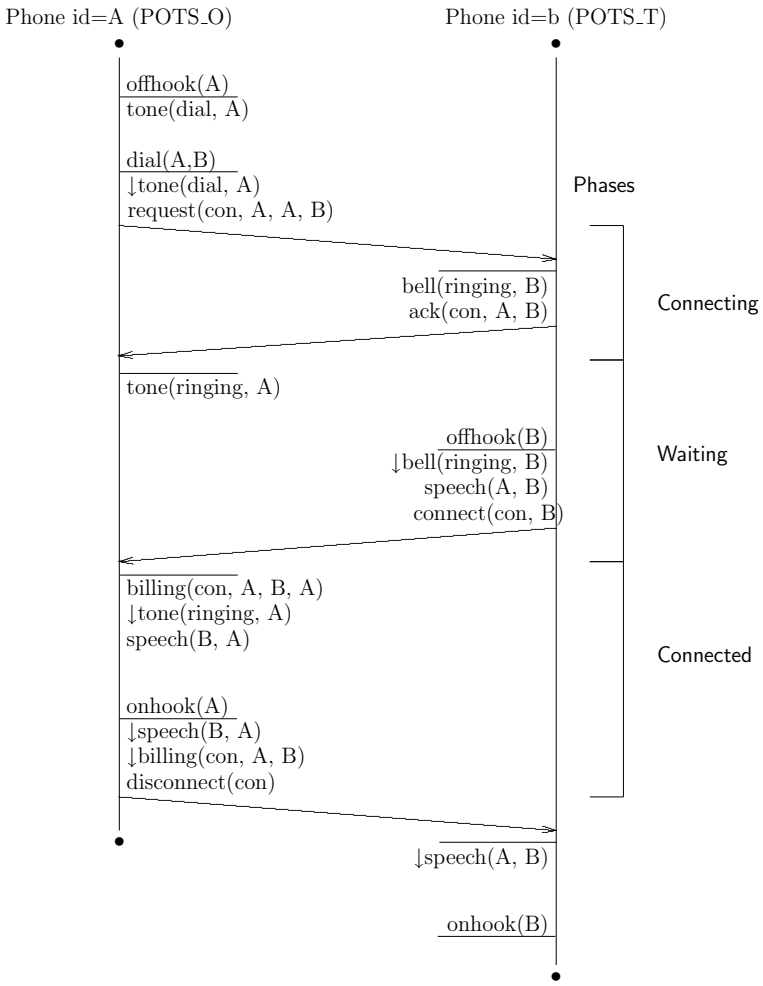

Figure 2.4: A normal POTS call.

The other control locations not used in the connection phases, 1, 3, and 6, are used to start and end connections.

Control Location 1 In this control location the feature waits for the user to start a connection attempt, but also accepts that the receiver is placed on hook, ending the connection.

Control Location 3 The control location is reached when the terminating party is busy (or for other reason decides not to accept the connection) and in such situations the originator gets a busy tone. The only behaviour POTS_O supports from this state is that the originator hangs up.

Control Location 6 The last control location is like control location 3, but here the terminating side hung up and all the originator can do is to hang up too. The difference between control locations 3 and 6 is that the originators phone has a busy tone in control location 6.

The Terminating Side of POTS (POTS_T)

Now that we have seen how the originating side of POTS works, we continue on to the terminating side shown in Figure 2.5, which we will call POTS_T.

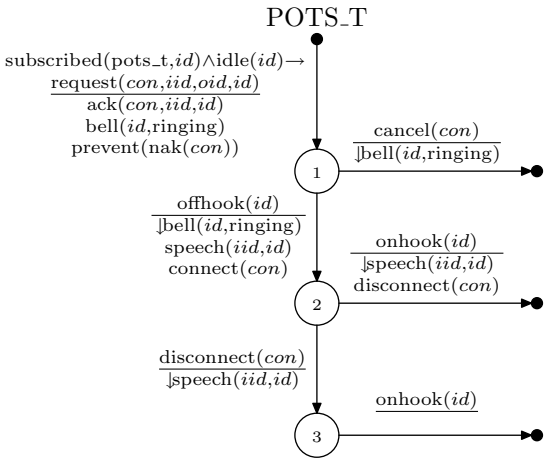

Figure 2.5: The terminating side of POTS.

The terminating side of POTS is the feature used by the callee to accept an incoming connection. We quickly walk through its control locations. If we look at Figure 2.4 we can note that events generated by POTS_T are used to end the connection phases.

Connecting Phase End The connecting phase ended and the waiting phase begins when a *request* event is observed. This takes the feature to control location 1 and starts ringing the bell. The transition also prevents the generation of a *nak* event. This is explained below.

Waiting Phase End The waiting phase ends and the connected phase begins when the terminating user lifts the phone $offhook$. This generates a *connect* event and takes the feature to control location 2.

Connected Phase End The connected phase is ended when the originating side disconnects, in which case the originating side will generate a *disconnect* event. This will stop the speech and take the feature to control location 3. If the feature observes that its phone, *id*, puts the receiver onhook, it will stop the voice and generate the disconnect event itself.

The other transitions, not described above, are for ending the feature in ways that make sense; assuring that the phone *id* reaches its initial state.

If we look at POTS_O and POTS_T in union, we can see that Figure 2.4 only describes one of several possible scenarios. The phones A and B interact with each other by sending messages to each other.

NAK

Two small and unexplained loose threads need to be tied up before the description of POTS is complete.

1. There is a strange event on the arc leading to control location 1 in POTS_T, a *prevent* event.
2. What happens if a phone has an instance of POTS_O or POTS_T running and a new connection attempt is made to that phone? Will the new attempt be accepted or rejected?

The idea we use in our model is to always try to reject all connection requests and this will be done using a special feature called NAK. By having a separate feature which always tries to reject connections we do not have to explicitly do this in, e.g., every control location of POTS_O and POTS_T. However, in situations where the POTS_T feature triggers and we actually want to override this behaviour we use a special mechanism. The *prevent* event is part of our model and used to give events priority over other events. Our framework has two kinds of priority. The first priority removes an event from the response (*prevent*s the event), and the second priority removes all response events from a processes responding with

a certain event (*block*s the transition). This mechanism is very general and is also used when, e.g., resolving interactions which often implies giving the behaviour of one feature priority over another behaviour.

The description of the small (auxiliary) NAK feature is found in Figure 2.6. The feature simply responds to all *request* events with *nak* events which, unless overridden, will inform the initiator of the connection attempt that the connection is rejected.

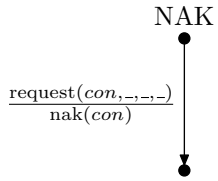

Figure 2.6: The NAK feature.

Now we have ensured that connection attempts will always get either an acknowledgement, *ack*, or a rejection, *nak*.

A new notation is also used in the NAK feature, the *request* event contains the symbol _ which represents the anonymous variable. This is an adoption from the Prolog programming language where the anonymous variable is used when the actual value of an argument will not be used and hence does not need be given a name. In the NAK feature the actual values of the initiator of the connection and originating and terminating side of the current leg will not be used, all that NAK needs to generate the correct *nak* event is the value of *con*. Hence, the other arguments are considered anonymous variables.

2.4.5 The Base System Revisited

We now make a further classification of components of the base system. Until now all components in the system have reacted to events they could observe, in other words all transitions this far has been reactive. This does however create a problem: what happens when there is nothing to react to? What makes a phone want to initiate a call? In the real world this is not a problem, the answer is simply that a human intervenes and generates stimuli, e.g., by lifting a receiver, pushing a button or performing another action to which the phones and subsequently the who system reacts. In our model we introduce *spontaneous transitions*.

Spontaneous Transitions Spontaneous transitions have no triggering event and can thus be taken spontaneously when the guard holds.

We can now define components of the system to be *active* or *reactive*.

Active component An active component contains spontaneous transitions and can thus generate events without responding to events.

Reactive component A reactive component has no spontaneous transitions and need to react to stimuli in order to generate events.

In the spirit of this classification, we define phones to be active. This, like in the real world, creates a telephony system that is driven by input from the users, via the phones.

Now that we know a little more about how we can describe components, we define the behaviour of phones in Figure 2.7.

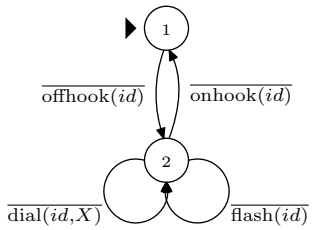

Figure 2.7: The active behaviour of a phone.

Note that there is no transition ending the phone automaton; phones are persistent and does not terminate like features.

The black triangle pointing at control location 1 in phone Figure indicates the control location where a phone starts, the state where the phone is on hook.

On hook In control location 1 a phone have its receiver on the hook. The only possible transition from this state is to lift the receiver $offhook$.

Off hook In control location 2 a phone have its receiver off the hook and can either *dial*, $flash$ or put the receiver *onhook*.

There also is another part to phones, the reactive part that keeps track of what happens to the state of the phone, e.g., the tone currently emitted. This part is described in Figure 2.8. This part reacts to the events generated by the active part.

It is assumed that the bell, tone or voice stopped also is emitted, otherwise and interaction is reported. This will be explained in Section 2.5.

The two parts of the phone are separated just for convenience, they could both have been present in the same process but we did not want the active components to have other dynamic variables than their current control location. The identity

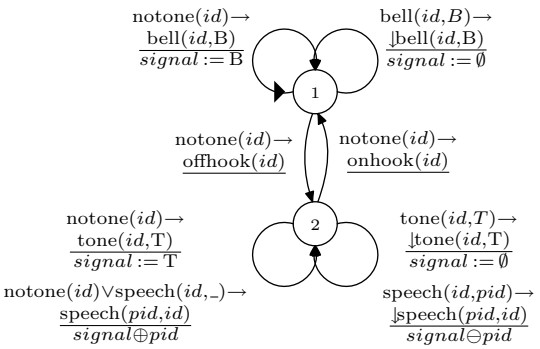

Figure 2.8: The reactive behaviour of a phone.

of the phone and any other variable data the active part has are all static. This is purely a design decision we made.

2.4.6 Features

A telephony system with only the POTS service does not have any interactions and the most serious problems in the system are probably the ones caused by hardware malfunctioning.

Using the way in which we have described features, as stand-alone pieces of functionality, it is obvious that new features can be added to the system as processes, i.e. an automaton for the feature is added and whenever the right conditions arise, the automaton will react and start executing. This is of major benefit when it comes to modelling the features since it avoids priority like problems which other kinds of feature (and feature) composition can rise.

Features are activated when the system state satisfies the activation conditions for the feature. The only problem that a new feature will have to take into account is that other features may also react to the same conditions and if the other features are incompatible they need to be blocked. In the examples above the blocking of incompatible behaviour (POTS) was done using the *prevent* and *block* events.

One advantage with our way to compose (and activate) features is that the maximal number of instances of a feature does not have to be given when a system is set up for analysis. The features are dynamically created and they are removed when they are finished.

Dynamic Activation With dynamic activation features exist only in the system when they are active. Feature are created when their activation conditions are met and they are removed when they terminate.

Static Activation With static activation Features are always present in the system whether active or not. Features wait in an idle or waiting state until activated and return to the same state when they are done, waiting to be activated again.

There are gains and drawbacks with the different methods. The biggest drawbacks with dynamic activation is that it is harder to analyse; it is easier to build methods for systems with a fixed number of components like with static activation will. However, with static activation the maximum number of simultaneous instances (e.g., forwardings from Fredrik) will have to be known beforehand, dynamic activation will create the instances when needed.

Now let us look at a feature to understand dynamic activation and how features are composed.

The service used in the *Example One* scenario was IN Call Forwarding. This service is defined by the combined behaviour of the IN Call Forwarding, Figure 2.10, and the Call Forwarding features, Figure 2.9.

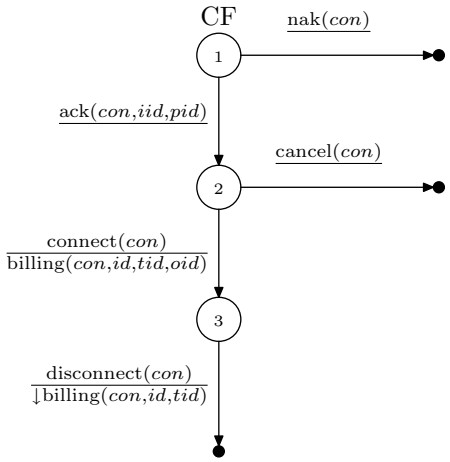

Figure 2.9: The Call Forwarding feature.

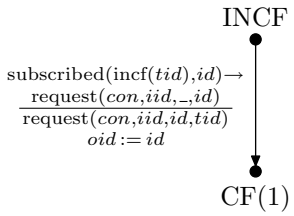

Figure 2.10: The IN Call Forwarding feature.

The IN Call Forwarding feature contains just the activation condition; all the real forwarding functionality is found in the Call Forwarding feature. The advantage

of this is that all forwarding services can use the feature to take care of the forwarding once the specific activation conditions have been met. Designing the Call Forwarding on Busy will only require a new small activation graph like the one for IN Call Forwarding, but triggering on a different condition.

We will look at the scenario in *Example One*, the forwarding example, and reason about it to see how the two kinds of activation work. Assume that we have static activation and that we start our system with two instances of forwarding from Fredrik's apartment to the summerhouse. This means that two calls can be forwarded at the same time. This is not a limitation, we could have designed the scenario we will look at so that there never occurs more than two forwarding instances[5]. When Peter calls, he will use one of the available forwarding instances leaving the other free to be used by another caller. If we have a scenario with more users, and more possible forwarding, the number of forward instances would just have to be increased to meet the new demands. Nevertheless, there is something wrong with this. Now look at the small scenario above again. We have two instances of forwarding and *both* will trigger and respond to the incoming call (leaving none left for further calls). From this, we conclude that static activation does not work with the current definition of how responses are generated. So why do we have this mandatory responding? Why not only have one component responding to a request? The answer is that we get another, perhaps trickier problem at or hands if we have processes synchronise in pairs. If we assume that only one feature responds to a request, it will be difficult to model situations where we want several features to react and respond to the same request. We want to detect feature interaction and one definition of interactions, as will be presented in Section 2.5, will be based on that inconsistent responses are generated.

Generators

We will now tie up a loose thread that was left dangling when defining how responses are generated. In the description of how responses, Section 2.4.4, we said that it was mandatory to respond for all components that could. We will use the concept of generators to tie this together with dynamic activation.

We use *generators* to assure that only one instance is created when the activation condition is met. Generators are responsible for creating the process for the feature by managing the triggering transitions of features.

In the IN Call Forwarding feature the generator manages the transition described in the IN Call Forwarding figure. For Call Forwarding there exists no generator and the feature relies on other features to start it.

[5]Note that we have not yet started any forwarding from the summerhouse to the apartment.

2.5 BASIC INTERACTIONS

The goal we set at the start was to describe a system in which interactions could be detected. What we have described so far is a model in which we can describe telephony systems as communicating components, but we have not given any method we can use for interaction detection.

Before we decide on how to detect the interactions, we need to know what we are looking for—what interactions *are*. In our three examples we said that we wanted to detect when features change the system's intended behaviour.

One important thing to understand is that not all interactions are undesired. Features are introduced to extend or restrict the behaviour of the base system and doing this means that they in some way will interact with the existing system. Hence, what we want to detect is *undesired interactions*.

The way we use to describe interactions is based on both how they manifest themselves and how they can be expressed. First, we remember that features describe a functionality that should be provided to the user. Failing to provide this functionality means that an undesired interaction has occurred. Feature interaction, i.e. that one feature subverts the operation of another feature, is however not the only reason for undesiredx behaviour. A badly designed feature can fail to implement its intended behaviour without intervention of other features.

2.5.1 Two Base System Properties

The first two types of interactions are used to describe properties of the base system.

The features manipulate the state of the system by generating events to features and the base system components. There are limitations on what the base system can do and whenever features want it to do something it cannot do at the moment, there is an interaction. The specific cases of this we will look at are interactions caused by *inapplicable events* or by *inconsistent events*.

Inapplicable Events

The base system components can only handle certain sequences of events. Trying to make a component violate this will result in an *inapplicable events* interaction.

Examples of inapplicable event are events that

- stops billing which has not been started,
- stops a tone that is not presently emitted by the phone, or

- starts a tone when another tone is being emitted (the emitted tone must be stopped first).

That something is wrong when a feature tries to stop billing that has not been started is obvious. That phones cannot emit two different tones at the same time is a restriction imposed by the way phones are expected to work rather than a hardware limitation. That tones must be explicitly stopped is a design choice taken when creating the model; the model could have been designed to implicitly stop tones when starting new ones.

Inapplicable Events Interaction An inapplicable event interaction occurs whenever the components of the base system cannot respond correctly to an event generated by a feature.

Inapplicable event interactions occurs when a feature has a faulty view of the state of the base system. Reasons for the faulty view can be either that the feature is badly designed or that another feature has modified the system in a, for this feature, unexpected way. Both these possibilities will have to be considered when solving this kind of interactions.

Inconsistent Events

The second type of interactions, *inconsistent events*, describes interactions that occur when features have different goals. The inconsistent events interactions are similar to the inapplicable events interaction in that they are detected by the events emitted by feature. However, inconsistent events describe a slightly different situation.

Features react on events by generating response events that should be applied to the system. An interaction occurs when two features react to the same event with events that can not be applied at the same time. Examples of events that are incompatible are events that

- start different tones on the same phone,
- try to place the next state of a connection in different phases, e.g., an *ack* event and a *nak* event, or
- have different phone identities in connections, i.e. requests from the same phone to two different phones.

Inconsistent Events Interaction An inconsistent event interaction occurs when a collection of response events contains two events that are inconsistent.

This flavour of interaction is separate from the previous type, inapplicable events, and since we here directly can identify the conflicting features. If one features tries to start a `ring` tone when another tries to start a `busy` tone there is no question about that these features are involved in the interaction. The feature causing the interaction when a tone cannot be stopped, as in an inapplicable event interaction, is not as easily identified since either the feature itself can be a badly designed or the interaction can be caused by another features modification of the system.

2.5.2 Feature Requirements

The two definitions of interactions presented so far detects only part of all possible interactions; none of the interaction types described so far can be used to detect the interactions found in the example scenarios. The examples does not contain interactions caused by the base system's inability to respond to events but rather the whole modelled telephony system's, base system and services alike, inability to meet user expectations.

As an example the two methods outline will not be able to detect the interaction we believe occur when billing is never started for a connection or the interaction in *Example Three* where it is not correctly ended.

Intentions and Requirements

When a feature is added to the system, it is added to provide the users with more functionality; there is an *intention* behind the service.

The intention of a feature can be described in words, e.g., the intention of the IN Call Forwarding feature in *Example One* is to forwarded all incoming calls to a predefined number.

Intention Functionality a feature is designed to implement defines the intentions of the feature.

Requirement A requirement is a formal description of an intention.

Just like we had to create a model for the telephony system, we will have to define a language in which to describe intentions. The language that will be used will describe requirements is logic.

Statements about properties of system components and behaviour is described using predicates like the ones introduces in the guards of the model, cf. Section 2.4.4 but now we need to be able to talk about time, e.g., we want to be able to express that a property should always be true. The language we use is the small bounded

time logic which allows us to make statements like that a property *always* should be true or that it only needs to be true *until* another property becomes true.

Example Base System Intentions

The base system components have intentions. Some of these intentions have been already been introduced as inapplicable and inconsistent events but there are intentions handling various system concepts that can not be captured by reasoning about the response events.

First, we introduce two predicates on the system state.

connection(con, A, B) which is true if there exists a connection with connection identifier *con* which connects the phones A and B.

billing(con, A, B, C) which is true if there exists billing, paid for by phone C, for the connection with connection identifier *con* between the phones A and B.

We can now formalise a requirement, BILL_1, for the intention that *there should exist billing for all connections.*

$$\Box(\mathit{connection}(con, A, B) \Rightarrow \mathit{billing}(con, A, A, B)) \qquad \text{(BILL\_1)}$$

Translated into English it would read that *"it is always true that if there is a connection there is also billing for it paid by the originator".*

The requirement BILL_1 has been written in a way that the telephony companies, who collect the bill, would like it; the requirement states that someone always pays for all connections in the system, however the requirement BILL_1 does not guarantee that billing is only present for real connections, i.e. there could be billing running without any connection. To capture this interaction, one of those who pay, another requirement is needed. This second intention, *whenever there exists billing, there exists a connection* is formally described in BILL_2.

$$\Box(\mathit{billing}(con, A, B, C) \Rightarrow \mathit{connection}(con, B, C)) \qquad \text{(BILL\_2)}$$

In English this would read that *"it is always the case that if there is billing there is also a connection."*. Together the two billing requirements BILL_1 and BILL_2 can check that billing works as intended. They are written in two flavours, BILL_1 is written to make sure that the initiator is the one paying for the connection. This is an example of an requirement in a system running only POTS. When connections are extended with legs, when forwarding occurs, the requirement will be broken and will need to be replaced with a requirement which knows about that forwarding will chance the structure of billing. The second requirement, BILL_2, is written in a manner that is ready for connection legs, it just states that if billing exists, there

is a connection to pay for. Further requirements are needed to make sure that only valid connections exist, but these requirements are not billing requirements but connection or switch requirements. Other connection intentions check that the connection phases, connecting, waiting and connected, are ordered in that order, i.e. that it is not possible to skip either of the two earlier phases when setting up a connection.

The interaction in *Example One*, the forwarding loop has to do with connections. The interaction can not be described as an inconsistent event interaction, there are never two conflicting events there's just no end to them, or an inapplicable event interaction, it is always possible to create another leg. One intention on connections which deals with this behaviour is that all connections end somewhere but this cannot be described with our temporal logic since it reasons about the existence of a property. Rampage forwarding will be captured by a progress property imposed on the searching, one stating that there should be no live locks.

Example Feature Intentions

In order to have a working set of intentions the set of base system intentions need to be complemented with the intentions of the features. In this section, we will sketch intentions and requirements for some of the example services.

The intention of Fredrik's IN Call Forwarding feature in *Example One* is that when in incoming call arrives to the subscriber they should be forwarded to another place.

First, we introduce two new predicates.

subscribed(A, S) which is true if phone A subscribes to the service S. The subscription can have client specific data D, e.g., who the a forwarding service forward and in these cases S is written $S(D)$.

calling(A, B) which is true if the phone A is currently calling phone B.

The IN Call Forwarding requirement can now be defined as:

$$\begin{aligned} &\mathit{subscribed}(B, \mathit{incf}(C)) \Rightarrow \\ &\quad \mathit{calling}(A, B)) \ \mathcal{U} \ \mathit{calling}(A, C) \end{aligned} \tag{INCF}$$

There are two instances of the requirement in *Example One*. In the subscription that forwards calls from the apartment to the summerhouse, Peter is A, Fredrik's apartment is B and Fredrik's summerhouse is C. In the subscription that forwards calls from the summerhouse to the apartment, Peter is A, Fredrik's summerhouse is B and Fredrik's apartment is C.

The intention of Malin's Account Card Calling[6] service is to let a specific account, not necessarily the standard one associated with the phone, be charged for the call.

$$subscribed(A, cc(C)) \Rightarrow \\ \quad \Box(billing(\_, C, A, B) \Rightarrow (pin\_ok(A, cc\_pin) \vee \neg has\_dialled(A, cc))) \qquad \text{(CC)}$$

Malin is A, the account (card) is C and the called party is B.

Another intention is that the user should be able to end the current call and begin a new one whenever the `hash` key is pressed during a call. The requirement checking this needs additional predicates but we will not show that here.

Malin's scenario also featured an answering machine and it has its own set of intentions. One of them is that all input is terminated with a `hash`, another that it should record incoming messages and be able to play them back later.

An example intention in *Example Three*, the scenario with 112, is that it should not cost anything being connected to the operator of the service.

The granularity of the model will decide which of the above intentions that actually can be captured in requirements. E.g., in order to formalise the answering machine intention that input is terminated with a `hash` the model needs to be able to observe when this specific key is pressed. Finding the appropriate level of abstraction is not easy [Cal98]. One important part of the model is therefore designing it so that the granularity can easily be changed. Adding mode abstractions, as well as removing them should be easy and not prompt remodelling of the whole system. This has been one of the goals when designing our framework.

2.6 DETECTING INTERACTIONS

We have now seen how telecommunication systems can be described, we have defined interactions, and we have also seen how features should be combined with the system to form a system with features. The part missing for the framework to be complete is how we detect interactions.

It is easy to visualise and simulate the operation of a telecommunication systems in the model that has been described in this chapter. There are several ways in which the parts of the framework can be used to find interactions and a method using exhaustive state space exploration was chosen since it is closely related to simulation. However, the number of possible states of the system model we have described grows exponentially with a parameter of the system, e.g., with the number of users and this is a problem for exhaustive state space exploration. Say that the number of possible states in a system with one user is five, in a system with

[6]This service is called Charge Call in the first interaction detection contest so we will assume that name in the name of the requirement.

two users is 92, in a system with three users is 1418 and in a system with four users is 2036. The number of possible states will quickly outgrow the resources of the computer trying to analyse the system and thus measures to reduce the effects of the state space explosion need to be taken for state space exploration to be a feasible method.

To understand the details of searching there are some things we need to know. We will first explain how searching is performed and then we will explain what scenarios are and how they are constructed.

2.6.1 Forward State Space Exploration

Assume that we have a system set-up containing information about the state of all the system components in the telecommunication system. We call this a *scenario*. Scenarios explicitly also contain initial data for the component like subscriptions and other, e.g., service related information.

Now assume that all system components have an initial state (control location) where they start executing[7]. The exploration will start from a state that is called the *initial system state*, where all system components are in their initial states, and the exploration will then create new *system states* by systematically generating and trying all events that can be generated in the state and the new states created by applying the responses to the events.

The terminology of system states is:

System State A system state describes an instantiation of all components in the system with the control location they *are* in and their data.

Initial System State The system state where all processes are in their initial state is called the initial state.

Quiescent System State A quiescent system state is a system state where there are no events waiting to be responded to.

The notion of quiescent system states will be used in deciding how to choose the next event to create a new state from the current state as detailed below. We assume that the initial system state is a quiescent state and note that initial states is what the scenario generator creates.

[7] Active components and components that never die, start in the state indicated with the black triangle in the component graphs and features are created on demand (by generators) when their activation conditions are met.

Intermediate System State System states that are not quiescent states are intermediate system states.

We use a wait-list to remember states we have yet to explore. Initially the wait-list contains the initial state. The exploration then repeatedly takes the first state in the wait-list and visits it to generate new quiescent states which are added to the wait-list and subsequently explored.

In short we start in the initial system state and then repeatedly collect the request events for all quiescent states we have not yet explored and add them to a queue of events to be tried. These events are used to generate new system states which end in new quiescent states. This is repeated until there are no more request events left or no more quiescent states to be explored. Detected interactions are reported and we also abort the current branch of the exploration since we do not want to explore state space where we know that something has already gone wrong.

2.7 RESOLVING INTERACTIONS

It is impossible to automatically resolve of intentional interactions [MaMa98] since a tool does not have all information to understand the combined intention and make a decision based on it, resolution demands manual attention and decisions. However, many interactions can be solved by assigning priority to either of the conflicting features.

A simple way of detecting and resolving interactions caused by bad feature design to just check the features individually with the base system and then tune them to work correctly. Sometimes the tuning will require that the feature is rewritten and at other times simple priority assignment needed. The hard part is identifying the service that changed the system state to one not expected by another service and this may require careful analysation of the trace of events leading to the interaction. Lima and Cavalli [LiCa98] actually proposes conformance testing which is done in several steps to ensure that the final system works correctly. Among the test steps is one step which tests the new feature in isolation, and one which tests the integration.

2.8 RESOLUTION OF THE EXAMPLE INTERACTIONS

This small section is just here for everyone who read the examples, got scared hastily browsed through the rest of the chapter accumulating fright, and now fears telephony systems and will hesitate if they lift a receiver. To stop this fear of being charged a million bucks for just using a phone we will present one way of resolving the example interactions.

Example One can be solved by limiting the number of hops a call may take. Every time a call is forwarded, regardless of where, the number of hops it has taken is

increased and when the number of hops reaches a predefined roof, the connection will be terminated by the switch. This solution is much simpler than the one where loops are actually detected and therefore used by today's systems. To detect loops the path the legs have taken needs to be recorded, the maximal numbers of hops solution only needs a counter.

Example Two is resolved by changing the behaviour of the `hash` key. The calling service waits for the key to be pressed for three seconds or more. Shorter signals will not be acted on and can therefore be used to operate the answering machine. This solution taken in today's telephony systems is to extend the interface of the phones to have keys for `hash`, the one you normally get when just pressing the key, and `long-hash` which you get when pressing the key for an extended time. This solves the interaction when the Account Card Calling service is redesigned to only react to `long-hash`es.

Example Three is something pulled out of my mind to cover aspects not present in the other examples but a possible solution to the interaction is to have the service handling the extra charging recognise situations where the charge is accepted by another party or by giving the 112 service the ability to identify and take over the air-time billing.

Chapter 3

Related Work

Feature interactions in telecommunication systems contains many interesting research problems. The approaches to handle the problems are categorised into three categories by Cameron and Velthuijsen [CaVe93].

Detection which aims to detect interactions in a system

Resolution which tries to resolve interactions (the interactions need to be detected first)

Avoidance which aims to prevent interactions occurring in a system

Further categorisation can subdivide the approaches into off-line and online approaches. Off-line approaches describe solutions used before the telecommunication system is taken into use and the online approaches are used in running systems.

Some of the problems, like interaction detection, have received a lot of attention from the research community while other problems, like interaction resolution, have received less. Other problems, like categorisation of interactions and categorisation of iterations causes, though not listed will also have to be addressed before an approach to handle the problem can be defined.

Regardless of the specific approach a model and one or more methods will have to be chosen. The model is the way in which the telecommunication system is described to the computer and the methods describes how to use the model.

Interaction Cause Categorisation

Cameron et al. [CGL+93, CGL+94] present causes of interactions in telecommunication systems. The causes are categorised into

Violation of Feature Assumptions Features can be designed with assumptions about naming of system components, data availability, administrative domains, call control and signalling. E.g. during connection establishment a phone can respond that is it busy or not busy. A service line Call Waiting can make this signalling ambiguous since it can treat a busy phone as idle. Services assuming idle to mean idle can have problems with this.

Limitations on Network Support Limitations can be in either equipment signalling or in the functionalities for communication among network components. An easy to understand limitation on equipment is the number of keys on standard phones. This limitation has lead to that some keys have been given different meanings in sometimes overlapping contexts.

Intrinsic Problems in Distributed Systems This category contains problems with personalised instantiations, timing and race conditions, distributed support of features and non-atomic operations.

The categories can be used to classify a detected interaction, but we find this is partly useful since some interactions can belong to more than one of the categories and no rules are given for how an interaction should be classified. However, the categorisation can be used to identify concepts and constructs in the telecommunication system that will cause interactions, which will be of great use to the system and feature designers.

In [CGL+93] Cameron et al. give a categorisation for interactions based on the number of components and users involved in the interaction. The first category presented is Single-User-Single-Component where only a single user and a single component is needed to produce the interaction. The other categories are Single-User-Multiple-Component, Multiple-User-Single-Component and Multiple-User-Multiple-Component. Though grouping interactions into the same category just because they can be created in systems of similar component structure the categorisation can be useful, e.g., in scenario generation or resolution. Knowing that an interaction is created by a single component can imply that solving it might not require modification outside that specific component. We find a weakness in this categorisation that an interaction can exist in several different categories, e.g., with one or more users, and that it in such cases is not specified which of the categories that should be used. This creates the possibility to assign an interaction to the *wrong* category.

The categorisations presented by Cameron and Velthuijsen [CaVe93] defines several views in which interactions can be sorted. The approach is versatile and recognises that interactions are likely to belong to different categories depending on the view taken. Categories to reflect different purposes are given and the goal of the categories is to separate interactions that can be resolved by applying old

methods from those requiring fundamentally new methods. The five different views presented are

The Software Life-Cycle View The phase during the development interactions can best be managed is used to classify the interactions.

The Network Configuration View Interactions are classified depending to the network configuration.

The Casual View The causes are used to classify interactions.

The Layered View The layers or the interfaces between layers where the interactions are detected are used to classify the interactions.

We believe that this categorisation is useful since the category an interaction belongs to is dependent on the view. The views can then be aid in looking at a collection of interactions, e.g., when trying to find the concepts in a model that causes interactions.

Other approaches that can be used for categorisation focus on the causes of interactions. Zygan-Maus [Z-M98] identifies two possible causes for feature interactions,

1. service level causes which are logical questions about the intended behaviour of services in different situations, and
2. technical level causes that are dependent on the implementation choices.

General methods for handling the interaction causes are also given:

1. service specifications need to be consistent and unambiguous;
2. network concepts, like switching functions and service control functions, need to be distinguished between; and
3. feature implementation concepts are needed, e.g. separation of concerns between service and network functions, and separate programs for service logics, resource handling and user profile handling.

We find these methods sound and they are present in our approach.

Wakahara et al. [WFK+93] presents a list of interaction causes more focused on how the interactions are detected, i.e. a casual categorisation. The indications used to detect interactions are duplication, redundancy, incorrect order of execution, inconsistency, vagueness/nondeterminism in processing and looping. This categorisation can be useful when designing a method for interaction detection.

An interaction in the inconsistency category can be detected with techniques looking for inconsistency the system. The authors also present detection methods for the different categories. The categories in this approach are good for deciding on how specific interactions should be resolved. We find the list of interaction causes to be most useful when defining a method for interaction detection.

In this thesis the cause of interactions is considered to be nondeterministic behaviour, deadlocks and inconsistency between different system components' view of the system state. Found interactions are categorised into categories determined by the concept violated, e.g., like *forwarding* or *billing*. Our categorisation is a blend of the different ones presented above.

Models of Telephony Systems and Detection

The description of the telephony system is the base for all research in feature interactions. The description allows or limits what the approaches can accomplish and also give the granularity that can be used. As mentioned in the previous Chapter, Cadler[Cal98] identifies finding the appropriate level of abstraction as very hard. Working with a sufficiently detailed model is of uttermost importance when working with interactions since a to high level of detail is likely to make detection in the model infeasible and a to low level will not be able to describe the interactions adequately.

The IN architecture [Q1200] defines a telephony system (AIN) consisting of three layers, the Global Functional Plane [Q1213] (GFP), the Distributed Functional Plane [Q1214] (DFP) and the physical plane. The GFP describes the system in a high level view with services and functional entities. The DFP is a more detailed description taking into account that the entities of the GFP may be distributed. The lowest level of the model, the physical plane, is concerned with signalling, protocols and the actual distributed nodes of the system. Today's approaches seem to focus on the GFP where it is easy to reason about individual services and their behaviour without having to consider the low level concepts of the physical plane or the added complexity introduced by the DFP.

Using the full description of the AIN is also commonly considered to complex or unnecessary since it is possible to reason about interactions in less complex models than full descriptions of AIN. However, Nyström and Jonsson [NyJo96] aims to formally describe the full AIN model in a detailed formulation with the primary aim to allow simulation of the model. We find that models with the complexity of the whole AIN are suited for simulation, not for analysis since the methods available today do not scale to the size of the whole AIN.

Lin et al. [LLG98] also uses the AIN but to detect interactions by identifying erroneous data sharing and feature disabling. A part of this work we find interesting,

and also use, is that it assumes that services are developed in isolation, without information about other services, and that apart from the interface of a service nothing is known about its design. Both these assumptions sets the work close to reality as this well captures the layout of today's situation on the telecommunication system market where third-party service developers create features.

The opposite approach is to define as little as possible of the telecommunication system and the services. An approach in this direction is described by Gibson [Gib97] who analyses service requirements to detect situations where the requirements conflict and hence services interact. However, using a description of the telephony system is more common.

Interesting approaches, models and notations used for features and telephony systems include:

State Transition Diagrams (STD), a graphical description technique, is used in an approach by Klein et al. [KPR97]. The notation is very similar to the one we use but the transitions are a more powerful when they allow transitions to *return values* as well as a postconditions (where the post conditions are comparable with our responses). However, the approach is weakened by the service composition described which fuses services into the basic call model making it grow in size and complexity with every service added. We do not find this method of composition feasible.

Rule-based service specification is adopted for the specification of service by Nakamura et al. [NKK98]. The transitions of the notation, called rules, have preconditions, an event and a postcondition. The transitions are however not given an order like in our framework and hence it can be hard to specify the intended order of the transitions without introducing special ordering predicates.

Use Case Maps (UCMs) by Buhr et al. [BAE+98, BEG+98] are used to defers detail when describing telephony systems. The authors identify one of the major problems when describing feature interaction in telephony as the lack of *first-way* descriptions, above the level of details. UCMs describe cases, situations that can occur in the system, using patterns for the different components. The originating party of a call has a specific pattern which contains stubs where feature functionality can be plugged in. Interactions are detected as inconsistent or unexpected behaviour for plug-ins.

Layered state-transition machines are used by Braithwaite and Atlee [BrAt94] to detect interactions. Higher layer machines describe enhancements of lower layer machines. Tokens representing different events in the system are passed through the layers and interactions can be found by looking at the reachable states of the compositions. Checking two features implies checking the

reachable states of all possible different layer constructions of the two features. The approach then analyses all possible combinations of composition to detect where the behaviour of the different compositions differ. Differences in behaviour between the compositions indicate that some kind of interaction occurs. The way we do composition only one composition is possible and hence we only need to analyse once.

Finite State Machines are used by Khoumsi [Kho97]. Detection is done by identifying undesirable states in the joint behaviour of services. The joint behaviour is created by merging the machines and resolution is accomplished by adding supervisors that force the machines to behave in an acceptable way. We believe that this method of resolution is of limited usage when the complexity of the system grows.

Coloured Petri-nets used by Cheung and Lu [ChLu95] and by Nakamura et al. [NKK97] has the big advantage that there exists many methods for analysation of Petri-nets, however there are still problems with any dynamic behaviour and this makes Petri-net approaches hard.

SMV, a model checker developed at Carnegie Mellon, is used by Plath and Ryan [PlRy98]. The general idea of the approach is to describe features formally as independent units of functionality which can be understood without detailed knowledge of the base system. The limited expressiveness of SMV limits the approach, as does the state explosion that occurs during analysation.

All of these notations have properties that make them suitable for some aspects of the problem of feature interactions but they also force issues like composition in directions we did not find suitable for our thinking which is the major reason why we introduced our own notation.

As an example of how feature composition can create problems, we look at the service definitions of the first feature interaction contest [GBG+98]. The presented the services described with a small textual description, giving an outline of the functionality, and a scenario extending the behaviour of POTS in a language called Chisel [AGG+98]. The scenario described all possible executions with the service but since it was given as an extension, rather than a stand alone component, it was sometimes hard to distinguish between the parts of the Chisel diagrams describing the actual service and the parts echoing POTS behaviour. Describing services using scenarios has the advantage that it is easy to follow different execution paths of the system. However, changing POTS would mean that all services would have to be rewritten to echo the changes in POTS. Service composition was accomplished by replacing or adding transitions to POTS. This works fine when only one service

is added but problems arise when two services are added[1]. The outcome of the composition of two (or more) services will depend on the order in which the different services are added to the system. Assuming that the first service replaces the same transition replaced by the second service the result will be that the second service will replace the first service's change and if unlucky the first service will never be activated. Adding them it reversed order would create a situation where the other service's change is replaced. Composition with replacement gives *priority* to the service added later to the system simply by the fact that they are added later. This may not be desired since it makes the composition order something that has to be taken into account when designing the model and system.

Another problem with this composition is that activation only is specified from the situation reachable from the initial POTS system, Situations where services like TWC, which actually mimics the behaviour of POTS, have recreated conditions like those found standard POTS will not be able to activate services since the substitution only is defined for POTS.

The second feature interaction contest [KMM+00] changed the description of services to be more like stand alone components. However, the method for service composition was the same. This creates a problem when adding more than one service to the system. Think about the situation where the system is in a particular state, e.g., where two phones are connected and two users speak to one another, and a service extends this state with added functionality. If the state is identified using the control location in which it is described, in our model that could be e.g., control location 5 in POTS_O, and not by the actual state of the whole system, e.g., a state which can be reached on observing a *connect* event, similar states added by services will not be recognised and extended with the same behaviour.

Scenario Selection

The way scenarios is chosen is important since it will influence the interactions detected and it is very important whenever the numbers of users adds complexity to the detection.

Kimbler [Kim97] introduces *Interaction Filtering* which reduces the number of problematic service combinations. Keck [Kec98] implements a prototype to identify these interaction-prone scenarios in IN services. Aspects, like resource usage, is used to determine if services are interaction prone.

Reiff [Rei00] presets an approach for scenario selection which is much like the one we use. Features are given an arity, denoting the number of users that is used in the features. When deciding the final number of users the set of common users is drawn from the sum of the arity of features in the scenario, much in the same way

[1] I.e. problems apart from the one that the all services echoes the behaviour of POTS.

we combine users. The approach is illustrated using a small set of features. Though mentioned, the presentation does not consider how feature data can change the arity of a feature.

Resolution

Otha and Harada [OtHa94] describe a telephony system using finite state machines and propose how interactions can be detected and resolved and resolution of interactions is done by priority assignment, a disjunction construct where there are competing rules and simple terminology corrections.

Zave [Zav95] reasons about what the intention of call forwarding services and then refines the services into more specific variants to resolve interactions. Some behaviour will be desirable in one instance of the forwarding services but undesirable in another.

Avoidance

Interaction avoidance has not received the same attention as other areas of the feature interaction problem. With avoidance the system is designed in ways that prevent interactions from occurring in the first place.

Zibman et al. [ZWR$^+$] present an architectural approach to minimising the number of interactions. The architecture uses agents for the different roles in a telecommunication system and separates concerns between them with the aim to reduce the number of assumptions in the architecture. The architecture separates the concerns of the roles users, services and the resources. One advantage of the separation is that the introduction of new services and new technology should not require modification of existing components.

Zave [Zav98] considers the real problem of feature interaction as a failure of modularity occurring when the specification of features or feature composition does not capture the intended behaviour. The strategy used, a pipes and filters architecture, improves the feature specification language and the rules for feature composition by structuring communication.

Hall [Hal98] presents a foreground/background model to avoid spurious interactions introduced by a naive straight merge approach to feature composition. Instead of operating on a single model, two models are used where the behaviour of the foreground model has precedence over the behaviour of the the background model. Spurious interactions are avoided by removing conflicting actions of the background model letting the foreground model continue searching undisturbed. *Real* interactions are still detected as they belong to the foreground model.

Chapter 4

Framework

This chapter will give a more detailed description of the framework we have designed to describe telecommunication systems and services. The definition here is more formal than the *vanilla* one presented in Chapter 2. More detailed definitions of the different concepts will be given. First the basic types and basic constructs will be introduced and then more complex concepts including our process model and the mechanism for composing processes to form an entire system model will be given.

4.1 VARIABLE DOMAINS

System components may have data variables. The variables are local and their types indicate the *domains* over which they range. The different variable types are presented in this section.

The assignment operator, $a := b$ which assigns the variable a the value b, and the check for equality, $a = b$ which is true if the value of a is equal to the value of b, are available for all variable types. Both assignment and equality checks can be done explicitly or implicitly. The implicit behaviour occurs when events trigger process steps and this works in the same way as matching in Erlang [AVW+96] or unification in Prolog[1], cf. [A-K91].

4.1.1 Phone Identities

All phones are given a unique identity which is used to reference phone (and sometimes, when convenient also the user behind the phone). Phone identities can

[1] A variable which has no assigned value will assume the value passed through an argument with its name. If the variable has a value when a value is passed in its name the value must be the same or else the matching or unification fails. Prolog, unlike Erlang, will associate an unbound argument with the local variable with the same name and assignment which can lead to that assignment occurs in the other direction, local variable to argument, by we do not use unassigned arguments so this will not occur in out framework.

1. be used in subscription information to denote the user who subscribes to a service,

2. be dialled as an ordinary telephone number when calling, and

3. be used as an account in services that, e.g., change the charging.

There is a potentially infinite number of phone identities but in each set-up of the system the number is fixed since there is a static number of phones in the scenarios and hence a fixed number of phone identities.

4.1.2 PINs

A Personal Identification Number (PIN) is used to identify a user for security reasons. The actual representation of PINs, e.g., if they consist of a predefined sequence of key presses, is not relevant to the framework.

In our system model, PINs are diallable and we give all services that require security one PIN number which represents that the correct PIN, whatever it might be, has been given. Any other input will be treated as a bad PIN, i.e. if a phone identity is given it will be considered a bad PIN. E.g., there is a PIN intl_pin for the INTL service, cf. Section 6.2. This makes the number of PINs in a given scenario fixed.

4.1.3 Tones, Bells and Voices

The tones and bells are different patterns of sound that are emitted by a phone. Tones and voices are emitted through the receiver when the phone is offhook, and bells from the external bell when it is onhook. There is a fixed set of possible values, shown in Table 4.1. The value $\emptyset$ indicates that there currently is no sound emitted from the phone (or that the set of connected voice paths to other phones is empty).

A common property of tones, bells and voices is that they are continuously active from the time they are started and remain active until they are stopped. Other signals that can be emitted by the phone, like announcements and the call waiting tone, are of a fixed length and hence need only be started and not stopped. Announcements are described in Section 4.1.4.

The set in `speech` contains the identities of the other phones currently connected with a speech path to the phone. In a normal POTS call this set contains the phone at the other end of the line, but services may add more identities, representing that there are several active voices, and the connected voices must therefore be a set rather than a single value.

Table 4.1: Tones, Bells and Voices.

Type	Description
-	$\emptyset$ (the phone is silent)
tone	`dial`
tone	`ringing`
tone	`busy`
speech	`speech`($\{id_1, \ldots, id_j\}$), where $id_i \in$ Phone Identities
bell	`ringing`, and special variants like `ringing(fast)`

4.1.4 Announcements

Announcements are used to relay information to a phone. The announcements are, as opposed to tones and bells, only started since they have a natural end when they have nothing more to relay.

Examples of announcements are fixed-length messages, e.g., strings such as ‘‘`Give PIN`’’ and ‘‘`Give Account`’’, or special tones like the `call_waiting` tone that is an announcement containing tones and not a voice message. The `call_waiting` tone is modelled as an announcement since it is applied on top of voices, something tones can not be, and since it is of limited length.

The number of announcements is fixed in a given system set-up.

4.1.5 Service and Feature Names

All services, and features, have their own unique names which are used in, e.g., subscription information.

The set of names is fixed in a given system set-up.

4.1.6 Control Locations

Features are composed of a finite set of control locations and transitions between these locations. The naming convention we use for the locations is the name of the feature followed by the node number in the graphical version of the feature, i.e. POTS_O(1) is the node denoted 1 in the POTS_O feature, cf. Figure 2.2.

Control locations are part of the process state.

4.1.7 Connection Identities

Elements of this domain are used to reference connections. Connections can consist of several legs, cf. 6.1.3 and there is no limit on the number of legs between any two phones. Connection identities are needed to identify specific connections and all the legs belonging to a connection.

New connection identities are created using the $fresh(con)$ function which returns a unique identity *con*.

4.2 EVENTS

Events are terms $e(d_1, \ldots, d_n)$, where e is an event type, taken from a finite set of event types, and where $d_1, \ldots, d_n$ are parameters, each of which ranges over a given domain. We use $\overline{d}$ to denote a tuple $d_1, \ldots, d_m$ of parameters.

Events are local or global. Local events only concern the component generating them like, e.g., the events for managing local variables like assigning them values. Global events are used as requests to update the state of other components, e.g., if a feature wants a phone to start ringing it generates the appropriate event to which the phone reacts.

4.3 PROCESSES

Processes are the computing elements in our model. The processes are used to represent all entities in the model that require a memory, i.e. they are used to represent the switch, the billing system, the phones and the features. There are processes for each of these entities and especially one for each phone and feature instance. If there are two forwarding features active from a user, there will be two processes, one for each of them.

The processes execute by taking transitions from their current control location to a new control location.

Each process has a set of typed *local variables*, each ranging over a given domain. The number of variables and their types depend on the process type. A feature process does not need the same variables as a switch.

A *process configuration* p is a pair $\langle \mathtt{c}, \sigma \rangle$ consisting of a control location $\mathtt{c}$ and the mapping σ from the local variables to values in their domains. Whether or not the value of a variable is defined in a certain process configuration is important for determining the possible behaviour of the process.

New processes can be created using the $create(pconf)$ event which adds a process with process configuration *pconf* to the global system state.

4.4 SYSTEM STATES

A global system state consists of a tuple $\langle \mathcal{P}, \mathcal{E}^s \rangle$ where $\mathcal{P}$ is a set of process configurations, $\{p_1, \ldots, p_n\}$, and $\mathcal{E}^s$ a set of events, $\{e_1, \ldots, e_n\}$. The process configurations represent all the active processes and the events are the events pending to be processed.

We say that a system state $\langle \mathcal{P}, \mathcal{E}^s \rangle$ is *quiescent* if the set of pending events is empty, i.e. $\mathcal{E}^s = \emptyset$.

4.5 TRANSITIONS

The dynamic behaviour of processes is given by a set of *transitions*. The form we draw transitions is shown in Figure 4.1.

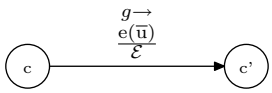

Figure 4.1: A normal transition.

In transitions

- $\mathtt{c}$ and $\mathtt{c}'$ are control locations,
- e($\overline{u}$) is the triggering event where e is the triggering event type and $\overline{u}$ is a tuple of variables,
- g is a guard, i.e. a Boolean expression, which must hold for the transition to be taken, and
- $\mathcal{E}$ is the set of response events that should be generated if the transition is taken.

Triggering is easiest described with an example. Assume that the event $offhook(\mathtt{A})$ is observed in the system, where $\mathtt{A}$ is a value belonging to the type of phone identities. Assume also that a feature process is in a control location from which a transition in which the guard g is $idle(id)$, the triggering event is $offhook(id)$ and the response is $\mathcal{E}$. The transition will be armed if the phone with the identity $\mathtt{A}$ is idle and it will be triggered for a process if the local variable id of that process is the same as that of the event, i.e. $\mathtt{A}$. If the transition is triggered the events $\mathcal{E}$ will be generated as a response to the request event.

A transition for creating a new process, $\langle \mathtt{c}', \sigma \rangle$, is shown in Figure 4.2. The transition creates a process which starts in control location $\mathtt{c}'$ with a mapping, σ, from the process variables, to the values of the event parameters, $\overline{u}$. $NAME$ denotes the name of the service or feature.

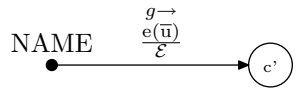

Figure 4.2: A transition creating a process.

A transition jump, Figure 4.3, to another graph with the feature called `NAME` and control location `C`, will continue the execution of the process in that graph with that name and in that control location.

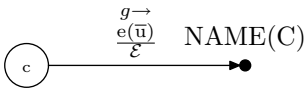

Figure 4.3: A jump transition.

A transition for destroying a processes is shown in Figure 4.4. Destroyed processes are removed from the system state.

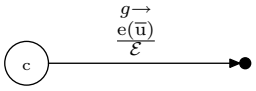

Figure 4.4: A transition destroying a process.

A control state independent transition, Figure 4.5, denotes a transition which does not involve any process configuration. Typically a feature that does not require a memory will consist of a single state independent transition.

A spontaneous transition, Figure 4.6, denote a transitions without any triggering event. A spontaneous transition can only be performed in a quiescent system state, and changes the state only of the process that performs it.

Like in Prolog we use the notation '_' to denote an anonymous variable. The anonymous variable is used when the value of a parameter will not be used. No two instances of anonymous variables are equal; the anonymous variable behaves as it does in Prolog.

Priority Events

All feature processes that react to an event will generate response events. It is clear that some of the events generated by one feature can be in conflict with those generated by another and in order to allow a feature to inhibit the response of other feature, we introduce two priority events.

prevent(e) prevents the event e from being generated; all occurrences of the event e is omitted from the responding events. All process steps containing the event e will still be taken, but without the prevented event.

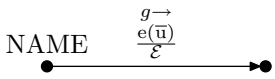

Figure 4.5: A control state independent transition.

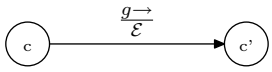

Figure 4.6: A spontaneous transition.

block(e) blocks any process whose response contains the event e. Process steps containing the event e can be perceived as if they had never been triggered.

Usage of the *prevent* event has already been seen in POTS_T where the *nak* event of the NAK feature is prevented when an incoming call, *request*, is observed. Had the *block* event been used instead the effect would have been the same for this small example but if the NAK feature would have had a behaviour that did not end directly, all the behaviour would have been blocked whereas only the *nak* event is prevented using the *prevent* event.

Note that we do not consider the option of letting an event with higher priority kill a process, but we allow processes to create new features. To explain this we say that a process A is *responsible* for a process B if A has blocked or prevented B, i.e. A is responsible that B will terminate without causing new interactions. Creating a new process requires that the creating process knows enough about the child processes to set the process state up using local variables. However, allowing a processes to kill another process would require the killer process to take over all responsibilities of the killed process, since the killed process will not be able to fulfill its responsibilities to other processes. This, however, implies that the killer process must know everything about the killed process and since we want to design features unaware of other features we do not allow this. Taking responsibility for a process only makes statements about the future, not the past.

We let $prio(\mathcal{E})$ denote the set of priority events in the set $\mathcal{E}$.

We define the operator $Set \oslash prio(Set)$ to be the operator which from the set Set removes all events of lower priority than an event in $prio(Set)$.

An anomalous situation is reported when there are circular priorities in the set of priorities, e.g., when a process A tries to block a process B which tries to block process A.

4.6 GLOBAL SYSTEM BEHAVIOUR

The intuition behind the global system behaviour is that all components that can respond to an event must respond, e.g., when a key is pressed it is possible that

several features generate responses. This will lead to situations like the one in the *Example Two* scenario where both the Account Card Calling service and the answering machine (service) responds to the request generated when Malin presses the `hash` key.

We say that a *process step*, $\langle \mathsf{c}, \sigma \rangle \xrightarrow{\mathrm{e}(\overline{d})/\mathcal{E}} \langle \mathsf{c}', \sigma' \rangle$, is taken in the system state $\langle \mathcal{P}, \mathcal{E}^s \rangle$ if

- the model contains a transition from c to c', labeled by $\frac{g \rightarrow}{\frac{\mathrm{e}(\overline{u})}{\mathcal{E}}}$;
- the triggering event $\mathrm{e}(\overline{d})$ is present in the pending events, $\mathcal{E}^s$;
- the set of process configurations $\mathcal{P}$ contains a process configuration $\langle \mathsf{c}, \sigma \rangle$ in which foreach u_i, $u_i \in \overline{u}$ is either undefined or $\sigma(u_i) = d_i$;
- σ' is σ extended with $\sigma'(u_i) = d_i$ for all $u_i \in \overline{u}$; and
- g is true in the context of σ' and $\mathcal{P}$.

If the process step is taken it will generate the events $\mathcal{E}$ as response and update the process $\langle \mathsf{c}, \sigma \rangle$ to $\langle \mathsf{c}', \sigma' \rangle$.

The process step can also be taken if either of the process configurations $\langle \mathsf{c}, \sigma \rangle$ and $\langle \mathsf{c}', \sigma' \rangle$ are the special element $\bullet$ denoting a destroyed or not yet created process.

Let $\langle \mathcal{P}, \mathcal{E}^s \rangle$ be a system state and let e be an event where $e \in \mathcal{E}^s$. We say that a set

$$\mathcal{Q} = \{q_1 \xrightarrow{e/\mathcal{E}_1} q_1', \quad \cdots \quad q_n \xrightarrow{e/\mathcal{E}_n} q_n'\}$$

of process steps triggered by e from $\mathcal{P}$, where each q_i and q_i' is either a process configuration or the special element $\bullet$, is a *maximal set of process steps triggered by* e if any process steps triggered by e from any process configuration $q \in (\mathcal{P} \setminus \{q_1, \ldots, q_n\})$ has lower priority than some process step in $\mathcal{Q}$. In other words, it is not possible to extend the set of process steps without including a process step which should be blocked according to the priority rules.

A *system step* $\langle \mathcal{P}, \mathcal{E}^s \rangle \xrightarrow{\mathcal{Q}} \langle \mathcal{P}', \mathcal{E}^{s\prime} \rangle$ can be taken whenever $\langle \mathcal{P}, \mathcal{E}^s \rangle$ is a system state, $e \in \mathcal{E}^s$, and

$$\mathcal{Q} = \{q_1 \xrightarrow{e/\mathcal{E}_1} q_1', \quad \cdots \quad q_n \xrightarrow{e/\mathcal{E}_n} q_n'\}$$

is a maximal set of process steps triggered by e from $\mathcal{P}$.

The next set of process configurations after the system step is

$$\mathcal{P}' = \{\mathcal{P} \setminus \bigcup_{q_i \neq \bullet} \{q_i\}\} \cup \{\bigcup_{q_i' \neq \bullet} \{q_i'\}\}$$

The set of events, which are not blocked by an event with higher priority, after the system step is

$$\mathcal{E}^{s\prime} = \{\mathcal{E}^s \setminus \{e\}\} \cup \{\bigcup_i \mathcal{E}_i \oslash prio(\bigcup_i \mathcal{E}_i)\}$$

Intuitively, a system step is triggered by an event e in the system state. All processes that can be triggered by e without violating the priority rules, perform process steps in which new events are generated. In addition, new processes are created if possible.

If $\langle \mathcal{P}, \emptyset \rangle$ is a quiescent system state and

$$\mathcal{Q} = \{q \xrightarrow{/\mathcal{E}} q'\}$$

is a spontaneous process step, where $q \in \mathcal{P}$, then $\langle \mathcal{P}, \emptyset \rangle \xrightarrow{\mathcal{Q}} \langle \mathcal{P}', \mathcal{E} \rangle$ is a *spontaneous system step.*

The next set of processes after a spontaneous system step is

$$\mathcal{P}' = \{\mathcal{P} \setminus \{q\}\} \cup \{q'\}$$

Note that spontaneous process steps are not synchronised with other process steps.

An execution of the system will consist of a sequence of system steps. Typically, each sequence of system steps will eventually result in a quiescent configuration. Upon reaching a quiescent configuration, a new spontaneous process step is chosen, which triggers a new sequence of intermediate system steps. The sequence of spontaneous events leading to a state is called the *trace* to the state.

In general, the behaviour of a system may be nondeterministic, due to different possibilities in the order of selection of the events to consume, and due to internal nondeterminism in any of the processes. However, we impose two rules on how the next event to be processed is chosen from the system state's pending events, $\mathcal{E}^s$, to prohibit events from overtaking other events in the order in which they are processed.

1. All the non-spontaneous events generated as the response to a spontaneous event will be chosen before a new spontaneous event is chosen. This will make the system process all *internal* events *before* looking for new *external* stimuli.

2. The *oldest* events in $\mathcal{E}^s$ are chosen *before younger* events. This will make the system behave as if it processes the whole response to a request before processing the response to the response.

Anomalous Behaviours

A sequence of system steps, spontaneous and non-spontaneous, which does not eventually reach a quiescent configuration is a *live-lock*, which is considered as an anomalous condition.

A quiescent state from which there are no spontaneous steps is identified as a *deadlock* which also is an anomalous condition.

A request event on which no process triggers, i.e. a false triggering event, is said to be an inapplicable event and is an anomalous condition.

Chapter 5

Feature Interaction Definition

As discussed in the introduction, an interaction is considered to occur when the composition of two or more features introduces some unexpected or anomalous behaviour. Furthermore, we find it convenient to distinguish two types of anomalies.

- Violation of Basic System Properties, which in its turn can occur if two features attempt to carry out conflicting events, or if some basic system functionality (typically related to billing, tones, or connections) is handled in an unsuitable manner.
- Violation of Service Requirements, which is check by formulating requirements on intended service functionality, and check that the intended functionality is satisfied when the service is composed with other features. A typical example is that the Terminating Call Screening service should not allow the establishment of a connection between certain pairs of users.

In this section, we present these different forms of interaction in the context of our model of the network.

5.1 INAPPLICABLE EVENTS

Features expect the system state to look a certain way and inapplicable events, cf. anomaly behaviours in Section 4.6, suggest that there is a mismatch between the actual system state and the system state the feature expects. This occurrence of inapplicable events indicate that there are interactions in the system.

For example, a feature, f, setting up a connection will expect that the connection and all associated information will be present until the connection is disconnected and deleted. If the connection is deleted without the knowledge of the feature there is an interaction between the feature f and the process that removed the

connection because f's attempt to disconnect the connection will result in an inapplicable event.

Inapplicable event interactions of this kind occur when the assumed usage of the base system components are violated.

It is not clear that the erroneous behaviour is created by the feature which fails to apply an event, but there the feature is the closest approximation to where the error can be found.

5.2 INCONSISTENCY BETWEEN EVENTS

The *inconsistent events interactions* are a special case of inapplicable event interactions. Since the framework allows us to analyse the set of responding events given to a request, we can use this to capture interacting inconsistent events before they become inapplicable events. Informally, we consider events to be inconsistent if they cannot be carried out simultaneously in a meaningful way or if there is a way. This method is useful for detecting interactions between features that perform different functions in some given situation than the detection of inapplicable events since it is easier to identify the processes (and hence features as they are represented by processes in the model) responding with the inconsistent events.

For example, if process, q_1, tries to start a `busy` tone in phone p at the same time that another process, q_2 wants to start a `dial` tone, it is clear the the processes q_1 and q_2 interact.

To detect interactions of this form, we must define events that are inconsistent and these definitions are presented with their components in Chapter 6.

5.3 VIOLATION OF REQUIREMENTS

Our third and final way to detect interactions is by defining requirements using temporal logic observers that report when a given behaviour is not met.

The reporting of inconsistent events and inapplicable events can, e.g., not detect that billing is terminated when the associated call is, they can only detect if the billing can not be performed as intended due to that either inconsistent event on the billing interface is requested or due to that the billing system is in an unexpected state.

To specify the requirements we allow some simple statements, Table 5.1, about events and the state of the system together with logic connectives, Table 5.2, and temporal operators, Table 5.3.

Predicates on the trace to a system state, about history, often has to be implemented as observers on the system state. Observers are auxiliary processes that monitors specific events and to keep track whether the predicate is true or not.

Table 5.1: Predicates.

Predicates are used to assert whether or not a specific statement about the system state or its trace holds or not. • Predicates can make statements and conditions on the state of the system, e.g., checking if a phone is idle, if a connection exists between to phones or if specific billing is present. • Predicates can make statements about the occurrence of events in the trace to the current system state, e.g., a predicate can check if an onhook has been generated.

Table 5.2: Logic connectives.

$\phi \wedge \psi$ true if both ϕ and ψ are true, else false.
$\phi \vee \psi$ true if either ϕ or ψ is true, else false.
$\phi \Rightarrow \psi$ false only if ϕ is true and ψ is false, else false.
$\neg\phi$ true if ϕ is false, else false.

There is one aspect of the system which makes the requirements handling problematic: loops. The system can by taking an event revisit a state, and this creates a loop in the exploration. If a requirement states that something will eventually hold and the exploration loops back the requirement may not be met. This lead to extensive reporting that requirements are not met so we removed the logical operators making statements about existence.

5.4 SCENARIO SELECTION

With the current method for exploration new states are introduced by the events that can be generated in the states. Reducing the number of events in the system will therefore reduce the state space explosion.

- The reactive part of phones can choose to do things, create events, hence the scenario generator tries to keep the number of phones down.

Table 5.3: Temporal logic operators.

$\Box\phi$ which stands for always ϕ meaning that ϕ should always (i.e. in all future system states) be true.
$\phi \ \mathcal{U} \ \psi$ which stands for ϕ unless ψ meaning that the logic formula ϕ should be true unless ψ becomes true.

- A user who can initiate calls, an originating side of a call, can generate new events in more situations than a user who can only accept calls, a terminating side; a caller can always pick up the receiver to start a call but someone waiting for a call will (in our model) only pick up when there is an incoming call. Both of the described uses are less powerful than a user who can both initiate calls and accept them. The scenario generator will try to limit the power of the users.

Keeping the total cost of a scenario down will help managing the state explosion problem, though there are other factors that will still need to be addressed to further reduce the state explosion.

To illustrate the scenario generation mechanism we will again use the example scenarios. Recall *Example One* where Peter tries to call Fredrik. To begin with, we will only have one forwarding in the system, the one form the apartment to the summerhouse (we will not have the one back again).

Scenarios with the IN Call Forwarding service *demands* four users in them, one originator, two terminators and a user who can be both originator and terminator and which also has the forwarding service. The scenario will also {ndefdemands information about to which of the three terminating phones the service is forwarded. Creating demands like this one needs a good insight of the service and all possible scenarios desired with the service. The reason why the demand for forwarding looks like this will be explained in following section.

Service Demands Services have demands on how scenarios must be set up in order to investigate the service. Demands can be on users, and available subscription data.

If we have the set-up for a scenario, we can just place the roles of Peter and Fredrik at the different phones and we have our example scenario. Name symmetries, like if it was Peter who had the service and Fredrik who called, are regarded as describing

the same scenario and are thus removed to keep the number of scenarios down. We can now create two scenarios with the one forwarding service and the three phones we have. For clearness, we call the phone with forwarding *Apartment*, the phone capable of accepting calls *Summerhouse* and the phone that can only initiate calls we call *Peter*. The difference between the two scenarios is where to the forwarding is done.

Scenario 1 *Apartment* forwards incoming calls to *Apartment*.

Scenario 2 *Apartment* forwards incoming calls to *Summerhouse*.

The scenario we have been looking at is Scenario 2, but the verify that the service works correctly the Scenario 1 will also have to be checked since there is the possibility that the service doesn't work as intended when it forwards calls to itself[1].

Can we be sure that the scenarios described are really cover all possible situations that can occur in our system? Actually, we know that they do not, e.g., they do not check cases with eight thousand users but due to the state explosion and the infinite number of possible scenarios we limit the ones we check to be those we can afford to check but which are still interesting and likely to cover the most interesting aspects of the services.

Scenarios we can check for IN Call Forwarding are like

1. what happens when the forwarded phone calls itself and is forwarded (either to itself or to another phone),
2. what happens when another phone calls the forwarded phone and is forwarded like above, and
3. does it matter whether the forwarded phone is idle or not.

Will this work if we have two services? Yes, in a similar manner. All services have demands that are simply combined to create scenarios. With two forwarding services, the scenario described in *Example One*, where both *Apartment* and *Summerhouse* can forward calls, the scenario generation becomes more complex and there are 19 different scenarios generated. Only one of these 19 possible scenarios exactly describe the situation described in *Example One*.

With two different services the demands of the services are combined in a similar way. Though the current implementation of the generator combines two services, is not limited to this. The generator can easily be modified to combine two scenarios

[1] When the forward service forwards to itself there is a loop which needs fewer components then the one used in *Example One*, but the scenario used in the example is more illustrative.

instead and this would enable it to generate scenarios containing any number of services.

One important thing with the scenario generator is that it is concerned with scenario set-ups only, it is not concerned with runtime behaviour of the generated scenarios. As a consequence of this the scenario generator does not know the actual number of connection legs and the number of instances of the services that may occur in a scenario, it only sets information about the base system and subscriptions to services. The scenario generator knows about the services and their initial system configurations, not the instances.

It is possible to extract *actors* from the definition of services. In a forwarding service there are three actors: an originator who calls the service subscriber who in turn forwards the call to a third party. Checking this forwarding service would hence require a minimal of three users where only one needs to be able to initiate a call, and the other two need only be able to answer to calls. Having the latter two be able to initiate calls would not aid in investigating the forwarding behaviour, it would just make the state space larger.

The actors places two kinds of demands on the scenarios, the number of actors and what kind of actor the service should be subscribed by.

Services can also need other resources than actors. One such resource is diallable numbers. If a service is activated when a specific number is dialled, the number must be present in the scenario.

A scenario is created by composing the demands of the participating services into a minimal set of demands that still satisfies all the original demands.

5.4.1 Exploration

For a specific scenario an initial configuration, the initial state, is constructed by selecting a set of services and the system components that the service set demands, cf. Section 5.4.

From the initial state the algorithm explores all possible executions. This is done by trying all possible events that can be generated in each state.

Exhaustive state space exploration is normally not feasible to use when dealing with complex systems[2] unless measures are taken to reduce the complexity. Our approach to reduce the explosion is to have a clever scenario generator and to use symmetry elimination. The scenario selection will generate as simple systems to

[2]Though there are indications that the problem can be managed cost effectively within budget as long as the method has been developed and the power of the computers continues to grow [GBM+99].

explore as possible and the symmetry elimination avoids exploring expanses of the state space which are renamings of other parts of the state space like in [NKK98].

Exploration Algorithm

We will now describe the implementation of the global system behaviour.

During the exploration we will use two queues, one containing states we still want to explore and one containing events which a quiescent state can generate. We add new elements at the beginning of the queues and also take elements from the beginning of the queues[3].

The global system behaviour, cf. Section 4.6, traverses one of all possible paths of execution from a given state. The exploration described here will traverse all possible paths.

We say that we visit a state when we collect response events that the active components can generate in it and add them to the event queue. When visiting a state we also record it[4] so that we can keep track of the states we have visited.

Exploration is performed in the following way:

1. Take the initial state and visit it by collecting the requesting events. Record the state and add collected requests to the queue.

2. If there is no requesting event left to be tried in the queue go to 3, else take the first requesting event not yet tried and apply it to the system and collect the response events, $\mathcal{E}$, the system generates and go to 4.

3. If there are no states in the state queue the exploration is finished and ended, else take the first state check if it has been visited. If it has been visited drop it and go to 3, else visit it by collecting the requesting events, $\mathcal{E}$. Record the state and add collected requests to the queue then go to 2.

4. If there are no responding events try the next request event (go back to 2).

5. Check that the response events don't interact. If they do, report this and try the next request event (go back to 2).

6. Use the responding events as requests on the system to create a new state and collect new responding events, $\mathcal{E}$.

[3]The queues is actually a stack as it operates in a Last In First Out manner. The reason for this is that this can be implemented more memory efficiently.

[4]Actually all symmetric states, with respect to renamings, are also recorded since this will further reduce the state space.

7. If there are responding events we have found an intermediate state and go to 5, else we have reached a quiescent state.

8. If the quiescent state has been visited before we stop exploring this path and go to 2, else we check that the requirements hold (don't indicate interactions), visit the state (record it and collect the request events) and continue from 2.

The exploration of a path is stopped and an interaction is reported if the following happens:

1. The set of responding events, $\mathcal{E}$, is inconsistent, i.e. there is an inconsistent events interaction.

2. All actions in the set $\mathcal{E}$ cannot be applied to the current system configuration, i.e. there is an inapplicable event interaction.

3. A requirement is violated in a quiescent system state.

It can be useful to report when a spurious event generates no response from the system in step 4 of the exploration algorithm. But, since this will generate a lot of events that are not responded to is it not always practical but can be useful during feature debugging. Phones can always generate, e.g., $flash$ events but these will more often than not fail to generate response from the services.

The algorithm uses depth first searching, so when a state which has already been explored is reached we know that either the system loops or that we have seen all states reachable from this state. In either case the exploration will not be continued in this path. However, if the state creates a loop the requirements are checked to make sure that they are not violated. This is not necessary if the state doesn't create a loop.

Only quiescent states are remembered, not intermediate ones. This optimisation is possible since since the exploration is deterministic and hence intermediate states always follow the same chain of events leading from the same quiescent state. It is not possible that it suddenly appears a new chain of events leading from a quiescent state already seen, then that quiescent state would not be the state seen.

When reporting interactions it can be useful to give the chain of events leading from the initial system state to the situation where the interaction occurs. We call such a chain of events for a *trace*.

Trace A trace is the chain of events leading from the initial system state to a system state.

The contents of the trace can be varied to allow different analysations of the trace. If the trace contains just the spurious events the trace illustrates how the users of the system can create the interaction but if the analysation aims at detecting the cause of the interaction all events, spurious and intermediate, are required.

5.5 RESOLUTION OF INTERACTIONS

The analysis method described in the preceding section will stop the exploration of an execution path when an interaction is encountered.

When an undesired interaction is found it will have to be resolved; the behaviour in the situation where the interaction is detected has to be specified. We use the trace to the interaction to identify the situation where the interaction occur. We then decide on how the interaction should be resolved. Interactions that are created by errors in the model, may requires that features are redesigned. This resolution method sould only be used as a last resort and only when other methods are to complicated to use. The more common way we use to solve interactions is through assignment of priority and this is best used for interactions caused by the generation of two inconsistent events. However, the actual assignment of priority is left to a user of the system since the system itself can not decide on the correct behaviour.

Priority can be applied either on the action level or on the transaction level. We will illustrate both levels with examples.

When IN Freephone Routing (INFR) changes the billed party of a connection by blocking the corresponding billing action of POTS on the action level, it still wants POTS to progress into the state where billing has been started. In our definition, INFR is not intended to replace all of POTS behaviour, like handling tones, it is only only to start a different for billing. INFR must ensure that POTS is able to handle the modification of the billing introduced by INFR, which in this case is trivial since the action that stops billing need not be aware of who is the billed party.

As an example of transaction level priority we look at the initial interaction between Charge Call (CC) and POTS. After POTS has responded to an $offhook$ event, it expects either a *dial* or *onhook* event from the user. If the number dialled is the number of the CC service, the intended behaviour of CC is not to establish a connection between the phone and the service number. Hence CC will block POTS from taking its entire transaction responding to the dial. However, CC takes its own transaction, leading to a state where the continued behaviour of the service is realised. The responsibility of CC will in this case be to make sure that POTS will progress from its current state where it expects an onhook or a dial (which is assured since CC restores a state similar to the one one it started preventing POTS in, or the call is aborted by the user hanging up).

Chapter 6

Case Study

This chapter is an illustration on a sample service set and how they are modelled using our framework. The service set is taken from the first feature interaction contest [GBG+98] which contains twelve services.

We will not use the base system and services of the second interaction contest [KMM+00] since the services in that contest often were limited versions of the services of the first contest, or contained more design errors.

6.1 A TELECOMMUNICATION BASE SYSTEM

For the purposes of this case study, we consider a telecommunications system to consist of phones, a switch containing a Service Control Point (SCP), and a billing system.

The processes in our model of a telecommunications system are

- one process for the switch,
- one process for the billing system,
- one process for each phone (user) in the network, and
- a set of dynamically created and destroyed processes that carry out the functionality of POTS and the active services.

The basic functionality of the switch, which maintains information about the current connections and about which Phones subscribe to which services, is not explicitly modelled as a process; we model this functionality by events that are generated by processes and which modify the status of connections. In the detection of interactions, we will formulate constraints on the allowed ordering of billing events, and on the allowed ordering of connection events.

In the following subsections, we describe the phones and the processes in the switch.

In our model we impose a limitation to how features may use events; a feature may at most change the status of one phone, and the phone is chosen once and for all. This restriction has been introduced because the somewhat unmanageable and confusing situation which easily rises when services freely change the status of any phone.

6.1.1 Phones

The phones represent the users in the model. The automata describing them is small with one control location for each state of the hook; one control location represents the phone with the hook on, and the other location represents the phone with the hook off. The transitions between, and within, the two states represent the actions a user could perform like lifting the hook or dialling a number.

The phone processes have spontaneous transitions which makes the exploration for interactions progress. As an example, the events for putting down the hook or dialling only requires that the phone is off hook. This behaviour is shown, again, in Figure 6.1.

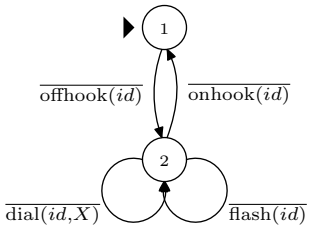

Figure 6.1: The active behaviour of a phone (again).

Phones also have a reactive behaviour which keeps track of the current state of the phones variables like if the bell is sounding or if there are voices realyed through the receiver. This behaviour is shown, again, in Figure 6.2.

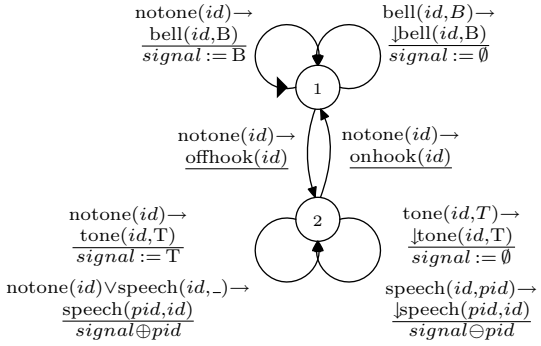

Figure 6.2: The reactive behaviour of a phone (again).

Phone Variables

Each phone is modelled by a process with a control location and whose local variables are:

id The identity of the phone, assuming values in the set of Phone Identities. This variable is always defined and never changed.

signal The tone, bell or voices that the phone emits or relays. The variable *signal* can assume values found in Table 4.1 where bells can only be emitted in control location 1 and non-tones in control location 2. Identities are added to the speech set using the special speech set operator $\oplus$ and removed using the operator $\ominus$. The speech operators make sure that the first phone identity added creates the speech set and the last one removed reverts the *signal* variable to $\emptyset$.

Initially, each phone is in control location 1, the variable *id* is the unique identity of this Phone, and the value of *signal* is $\emptyset$.

Predicates

The predicates on phones are:

$idle(id)$ is true if the phone *id* is in its initial state, i.e. in the control location 1 with *signal* set to $\emptyset$.

$notone(id)$ is true if the *signal* variable of phone *id* currently is $\emptyset$.

$bell(id, bell)$ is true if the *signal* variable of phone *id* currently is *bell*

$tone(id, tone)$ is true if the *signal* variable of phone *id* currently is *tone*.

$speech(id, pid)$ is true if the *signal* variable of phone *id* currently is a speech set which contains the phone identity *pid*, i.e. that *id* is speaking to *pid*.

Spontaneous Events

Phones have four spontaneous transitions which can generate events:

$offhook(id)$ can be generated by phone *id* when in control location 1 and the phone will move to control location 2.

$onhook(id)$ can be generated by phone *id* when in control location 2 and the phone will move to control location 1.

$dial(id, id')$ for any $id' \in$ Phone Identities can be generated by phone id when in control location 2 and the phone will remain in the same control location.

$flash(id)$ can be generated by phone id when in control location 2 and the phone remain in the same control location.

The structure of these transitions is shown in Figure 2.7. Here and in the following, we adopt the convention that in graphical descriptions of transitions, the first parameter to an event is omitted since it is always identical to the value of the local variable id.

Reactive Events

In addition to the change of control state, the value of the variable *signal* in a phone can be changed directly by an event of another process by generating the event for it. The events change the value of *signal* inside a phone with identity id are the following.

$tone(id, tone)$ The *signal* is set to the value *tone*, where $tone \in$ T*ones*. It is assumed that the phone is in control location 2.

$\not\downarrow tone(id, tone)$ Reset *signal* to $\emptyset$ assuming that the phone is in control location 2 and that *signal* has value $tone \in$ Tones.

$bell(id, bell)$ Set *signal* to the value of *bell*, where $bell \in$ Bells. It is assumed that the phone is in control location 1.

$\not\downarrow bell(id, bell)$ Resets the bell tone to $\emptyset$ assuming that the user is in control location 1 and that *signal* has the value where $bell \in$ Bells.

In addition, there are other events, as follows.

$announce(id,$ `message`$)$ Announce the message ‘‘`message`’’ to the phone id and then leave the state of the phone as before. Announcements assume that the phone is in control location 2.

$announce(id,$ `call_waiting`$)$ Emit a call-waiting tone to the phone id, assumed to be in control location 2, and thereafter leave the state of the phone as before.

$display\_number(id, id')$ Display the callers identity id' to the called user, id. We do not model any effect on the local state of the phone; the local state of the phone is left as before. The phone can be in any control location.

An interaction is reported if the assumptions are not met since this indicates that a service expects the state of the phone to be something else than what it is.

Inconsistent Phone State Events

The limitations causing interactions are either physical or logical (or both) in the sense that a normal phone cannot emit two bell signals at the same time, or should not do it. Inconsistent events interactions fo phones are:

1. Two different tones cannot be started at the same time since phones only can relay one tone at a time. Pairs of events $tone(tone_a)$ and $tone(tone_b)$ are inconsistent when $tone_a$ is different from $tone_b$.

2. The pair of events $tone(tone)$ and $\downarrow tone(tone)$ is inconsistent for any tone $tone$, since a tone should not be started and stopped at the same time.

6.1.2 Billing System

We model billing which can distinguish between multiple connections between the same parties using the connection identities. The billing system is shown in Figure 6.3.

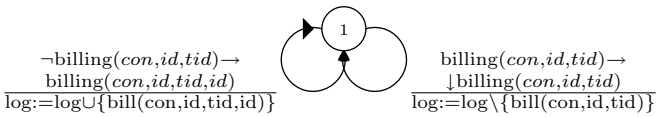

Figure 6.3: The Billing.

Billing System Variables

The billing system is modelled by a process with a control location and whose local variable is:

log The current billing is recorded in the varible *log* set. Each entry (connection leg) for which billing is currently active is represented by the entry $bill(con, oid, tid, payid)$ where *con* is the connection identifier, *oid*, *tid* and *payid* are the originating, terminating and paying parties.

Predicates

The predicates on the billing system is:

$billing(con, oid, tid)$ is true if the *log* variable of the billing system contains the entry $bill(con, oid, tid, \_)$.

Events

Billing is handled by events of the billing interface which begins and ends billing.

$billing(con, oid, tid, payid)$ Start billing in connection *con* for the leg between *oid* and *tid* paid by *payid*. This event assumes that no billing is already present for this leg.

$\downarrow billing(con, oid, tid, end\_time)$ End billing in connection *con* for the leg between *oid* and *tid*. Parameters as when starting billing. The event assumes that the billing has been started.

Errors are generated if the assumptions are not met.

The intended behaviour of the billing, for the providers viewpoint, is that all active connections are paid for and that and, from the subscribers viewpoint, that you only pay for connections you should pay for and that no legs are double charged, i.e. that you don't pay twice.

Inconsistent Billing Events

The billing we use in our model has one restriction on what we want to allow.

1. Billing may not be done twice for the same connection leg.

6.1.3 Switch

The behaviour of the Switch is modelled as a collection of processes, each of which defines a certain functionality. There are two special types of processes, the originating and terminating sides of POTS, which represent the basic mechanism of call handling. Additional services are represented by additional processes. In an initial state, no processes are present in the Switch. Processes are created when the functionality which they represent is invoked.

Switch Variables

Before presenting any process, it will be helpful to present how we model connections of the Switching system. At any point in time, the Switch maintains a set of connections. Each connection has a unique identity, usually denoted *con*. A connection is a finite sequence of connected *legs*. Each leg has an originating and a terminating phone, such that the terminating phone in one leg is the originating phone in the next leg. Thus, a connection is characterised by a unique identity

con and a sequence $leg(id_1, id_2)\ leg(id_2, id_3)\ \cdots\ leg(id_{n-1}, id_n)$ of pairs, each of which represents a leg in the connection. This is illustrated in Figure 6.4. The figure also contains an *ack* event which terminates the connecting phase.

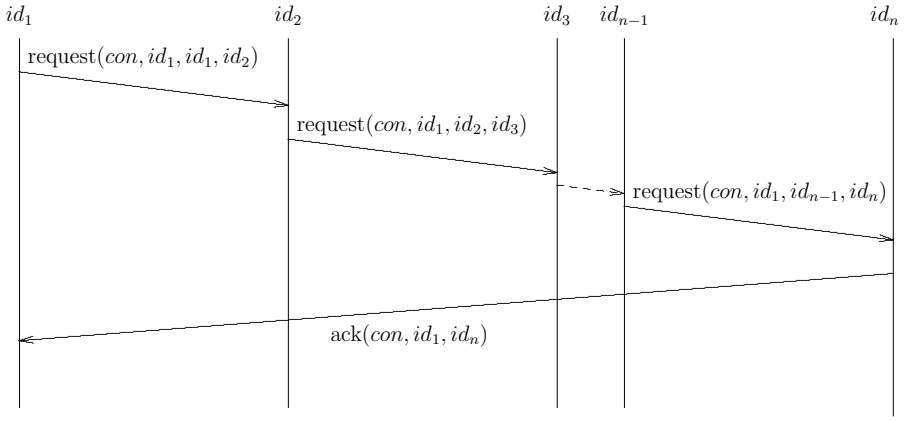

Figure 6.4: Connection legs.

The switch also holds a set of service data to keep track of subscriptions and related information.

Predicates

The predicates on the switch are:

subscribed$(service, id)$ is true if the phone *id* subscribes to the service *service*.

engaged$(servcie, id)$ is true if phone *id* has the service *service* engaged. Engagement is accomplished using the *engage* event.

connection(con, iid, pid) is true if there exists a connection with connection identity *con* between the phones *iid* and *pid*.

Service Events

The events managing services in the switch are:

engage$(service, id)$ Inform the switch that the *service* service has been activated for phone *id*.

$\downarrow$*engage*$(service, id)$ Inform the switch that the *service* service is not active for phone *id*.

Connection Events

The events to handle connections in the switch are:

request(con, iid, oid, tid) Request to establish a connection leg between an originator *oid* and a terminator *tid* in the connection *con* that was originally initiated by the *iid*.

cancel(con) The originating caller retracts her request for establishing a connection, *con*, (usually by performing an *onhook* event) before the callee has picked up the receiver, i.e. before the terminating side has answered the connection request with an *ack* or a *nak* event.

ack(con, iid, pid) Terminate the set-up of a connection, by accepting that connection *con* from *iid* to *pid* can be established. It does however not establish the connection.

nak(con) The terminating side refuses to establish the connection *con*.

connect(con) A phone accepts the connection *con* i.e. the callee performs $offhook$ event, i.e. after the *ack* event.

disconnect(con) Generated either by the caller or callee after a *connect* event, and causes all legs of the connection *con* to be destroyed.

The switch monitors the connection requests, cancellations, connects and disconnects in order to maintain information about the currently active connections. The requests are monitored in order to capture all legs of the connection (since this information is not present in the final connect event).

It is possible for events in the different phases to block other events in order not to enter the next phase, e.g., a *nak* can be given higher priority than a *request* or vice versa, whichever implements the intended behaviour of the service with the higher priority. Priorities are assigned using the priority events in Section 4.5.

Inconsistent Connection Handling Events

The life of a connection was divided into three distinct phases, the first ending when the terminating side answers the request, the second ending either when the connection is cancelled or connected, and the third ending when the connection is disconnected. Interactions in the connection set-up occurs when processes try to either direct the connection to different terminating parties, or when not all processes try to enter the next phase of the set-up at the same time.

As an example of the first type the pair of events $ack(con, iid, pid)$ and $nak(con)$ is inconsistent, for any connection con and any users iid and pid since the event $ack(con, iid, pid)$ allows that a connection might be established whereas the event $nak(con)$ is a refusal to establish the connection.

The pair of events $ack(con, oid, tid)$ and $request(con, iid, oid, tid)$ exemplify a situation of the second kind, i.e. one of the events wants to terminate the set-up of the connection (remain in the first phase) whereas the request event intends to further extend the set-up of the connection to another user (enter the second phase. The situation is the same if the ack is substituted for a nak.

6.1.4 Services

Services are reactive processes. They are not themselves able to generate events but have use switching functionality to accomplish this. As an example they must create processes which in turn send connection events.

Service Variables

The services have a set of local variables which are the following:

id the identity of the phone that subscribes to this activation of the service containing the process,

con The identity of the connection in which this process is active.

iid The caller that originally started the connection (the initiator) in which the process is active,

pid The callee that is the final destination for the current connection, the connection party, in which the process is engaged, though the value can be changed during the set-up of a connection, e.g., with call forwarding,

oid The originating phone in the connection leg in which the process is defined, and

tid The terminating phone in the connection leg in which the process is defined.

The services also have a spare set-up of all local variables but the *id* so that they can implement a bridge. A bridge is a facility which allows connections to be put on hold while another connection is active. This set of variables have the same name as the normal ones marked with a $'^b'$, e.g., the connection number, *con*, on the bridge is called con^b.

The bridge is modified with a special set of events:

clrbridge(*id*) Clear the bridge variables for phone *id*.

toggle(*id*) Switch the set of normal variables with the set of bridge varieables for phone *id*.

Auxiliary variables, like global subscription information, is stored in the switch.

6.2 SERVICE DESCRIPTIONS

Plain Old Telephone System (POTS)

We first present how we model the basic functionality involved in POTS. This functionality is modelled by two kinds of processes: POTS_O at the originating side and POTS_T at the terminating side. In addition, there is a transition which is not connected to any process state, called NAK.

The control locations of POTS_O and POTS_T are shown in Figures 6.5 and 6.6.

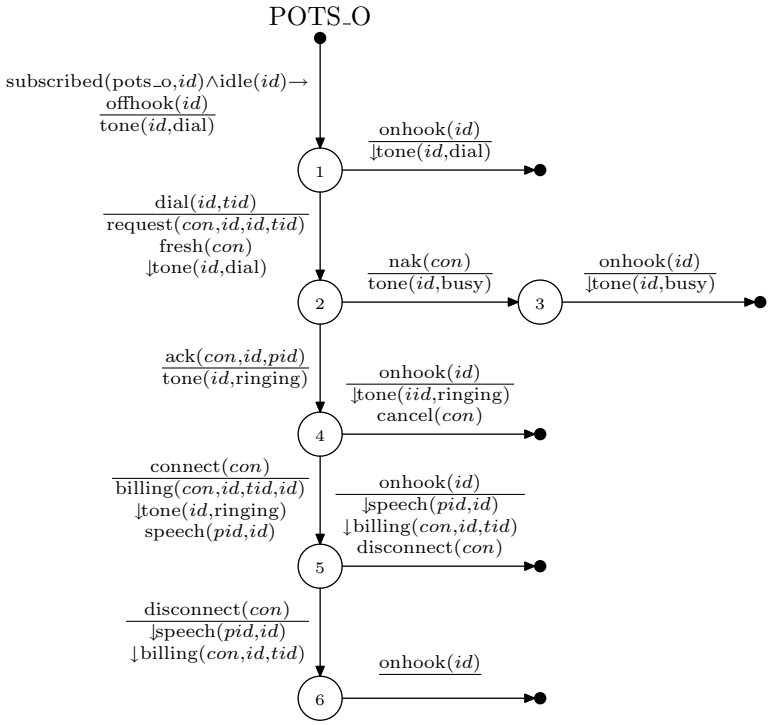

Figure 6.5: The originating side of POTS (again).

The POTS services does not use the bridge variables.

Below follows a description of how the two processes in a call leg collaborate in call handling.

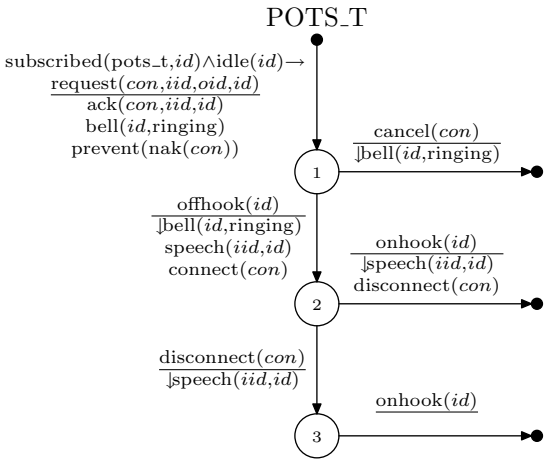

Figure 6.6: The terminating side of POTS (again).

1. When phone *id* has subscribed to POTS_O, and is in control location 1, an *offhook* event creates a process POTS_O. The new process will get the local variables *id* and *oid* from the phone that generated the *offhook* event. The value of the local variable *id* will be assigned in the POTS_O process that is created. In order to make the value of *oid* equal to that of *id*, the assignment $oid := id$ is included among the events. The event $tone(id, dial)$ gives a dial tone to the now active phone.

2. Having a *dial* tone, a phone *id* can generate an event $dial(id, tid)$ where *tid* is the identity of a terminating phone for the leg that will be set up. This triggers POTS_O to move from control location 1 to control location fignode-name2, stop the `dial` tone, and send a *request* event to set up a connection with a unique identifier *con* from *oid* to *tid*. The connection number is generated using *fresh* event.

3. If the phone *id* (which is the *tid* in POTS_O) is *idle*, the *request* event triggers the creation of a POTS_T process (in Figure 2.2) in control location 1. The creation is performed in the arc to control location 1, which also starts the ringing bell at phone *id*, and returns an *ack* to the *iid*w which originated the connection.

4. When POTS_O receives the *ack* event, it moves to control location 4 and starts the `ringing` tone.

5. When the terminating phone performs an *offhook*, it triggers the arc from control location 1 to control location 2 and generates a *connect* event.

6. The *connect* event causes the originating POTS to stop the `ringing` tone and to establish a *speech* connection between the two parties. It also generates a

billing event, which says that it (*oid*) will pay for the connection leg of *con* between *oid* and *tid*.

7. If the originating phone waits for the terminating side to pick up, i.e. is in control location 4, it can retract the connection request with an on hook, which sends a *cancel* event to the terminating side. On receiving a *cancel* the terminating side simply stops the bell.

8. At (almost) any time, the connection can be torn down by an *onhook* event, which causes one POTS to send a *disconnect* to the other POTS, after which the two state machines die.

The POTS_T process only handles the case that the callee is idle and free to receive the incoming call. In case that the callee is not free, an incoming *request* should trigger a *nak* event to notify the caller that the callee is busy. This is performed by the transition in the diagram NAK. However, the transition which generates $nak(con)$ has lower priority than any transition which generates an event $ack(con, iid, pid)$ for the same connection *con*. In this way, the *nak* event is generated if and only if the callee is not free to receive the incoming call.

Call Forwarding Busy Line (CFBL)

The Call Forwarding Busy Line service forwards all incoming call when the subscriber is busy.

The definition of the service is shown in Figure 6.7.

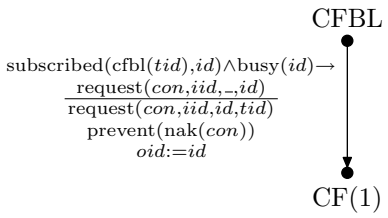

Figure 6.7: The CFBL service.

The subscriber is associated with a number, *tid*, to which the service forwards calls when activated.

The service is armed when it is subscribed and the subscriber is busy.

The service is activated when it is armed and a *request* event to the subscriber is observed. Upon activation it will extend the incoming connection with a leg to *tid* and prevent the standard POTS behaviour to nak busy terminators.

The service is a forwarding service and requires the CF feature, cf. 6.2.1, which handles the service behaviour after activation.

The service operates on the terminating side. When testing the service an originator, making a call to the busy subscriber, is needed as well as two terminators to whom the call is forwarded.

The service will when activated interact with other services trying to modify the connection behaviour at that point.

Calling Number Delivery (CND)

The Calling Number Delivery service displays to the subscriber, using a display, the identity of all incoming calls that terminate at the subscriber.

The definition of the service is displayed in Figure 6.8.

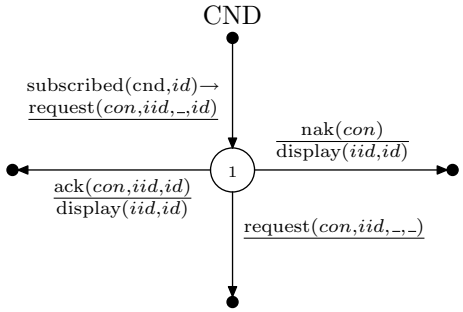

Figure 6.8: The CND service.

The service is armed when it is subscribed.

The service is activated when it is armed and a *request* event to the subscriber is observed. When activated the service will wait to observe if the call terminates at the subscriber, i.e. that the call is not forwarded. Only if the call terminates the identity of the incoming call will be displayed.

The service requires that the subscriber is able to display numbers.

The service operates on the terminating side. When testing the service an originator, making a call to the subscriber is needed.

IN Freephone Billing (INFB)

The IN Freephone Billing service allows the subscriber to take over the charge for the last leg of the connection in which the subscriber is a terminating party.

The definition of the service is shown in Figure 6.9.

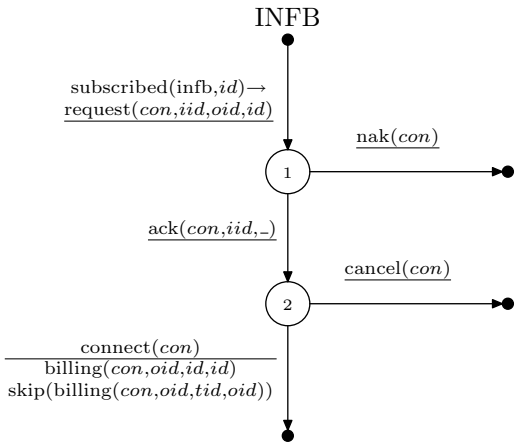

Figure 6.9: The INFB service.

The service is armed when it is subscribed.

The service is activated when it is armed and a *request* event to the subscriber is observed. If the request leads to a connection, i.e. that an *ack* and a subsequent *connect* are observed, INFB will prevent the billing POTS starts and start one of its own instead.

The service relays on that the normal POTS billing is terminated.

The service operates on the terminating side. When testing the service an originator, making a call to the subscriber is needed.

The service interacts with services that modify the billing of the same leg.

IN Freephone Routing (INFR)

The IN Freephone Routing service is intended to forward incoming calls. The actual forwarding is decided when the call arrives using special functionality in the switch. The decision can be based on, e.g., the time or by information about the caller.

The service definition is displayed in Figure 6.10.

The subscriber is associated with a number, *tid*, to which the service forwards calls when activated.

The service is armed when it is subscribed.

The service is activated when it is armed and a *request* event to the subscriber is observed. Upon activation it will extend the incoming connection with a leg to *tid* and prevent the standard POTS behaviour to nak busy terminators.

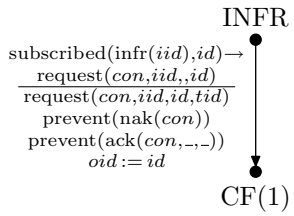

Figure 6.10: The INFR service.

The service is a forwarding service and requires the CF feature, cf. 6.2.1, which handles the service behaviour after activation.

The service operates on the terminating side. When testing the service an originator, making a call to the busy subscriber, is needed as well as two terminators to whom the call is forwarded.

The service will when activated interact with other services trying to modify the connection behaviour at that point.

In the model the number to which calls are forwarded, *tid*, is fixed so there is no extra functionality for making the decision on where to forward.

IN Teen Line (INTL)

While within a given time interval the IN Teen Line service requires the subscriber to dial a valid PIN before outgoing calls can be originated.

The service definition is displayed in Figure 6.11.

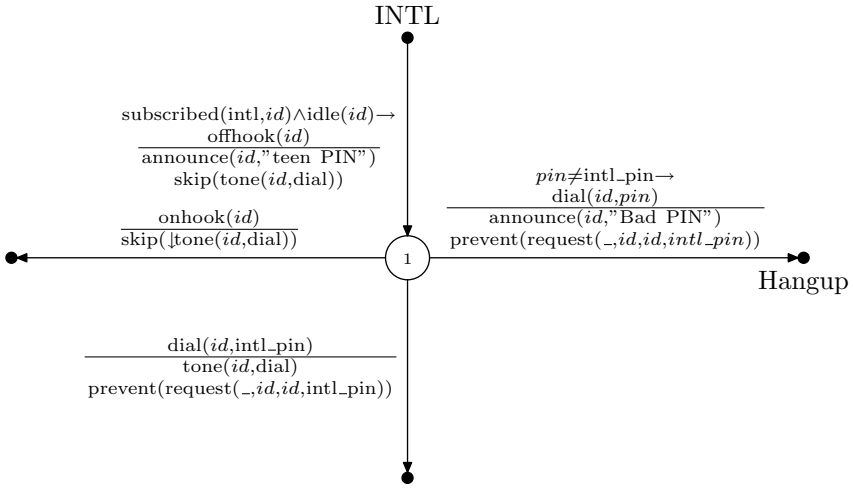

Figure 6.11: The INTL service.

The subscriber is associated with a PIN which will have to be given before the service allows calls to be originated.

The service is armed when it is subscribed and the subscriber is idle.

The service is activated when it is armed and an *offhook* event by the subscriber is observed. When activated it will require the subscriber to dial a PIN before outgoing calls can be originated. If the correct PIN is dialled the subscriber will be allowed to place outgoing calls. If the correct PIN is not dialled. i.e. a bad PIN is dialled, the subscriber will not be able to originate calls and will be informed to hang up. The normal POTS behaviour will be prevented when the service is active.

The service require the HANGUP feature as this service is used when a bad PIN has been dialled. The service also requires an announcement functionality so that it can alert the subscriber.

The service operates on the originating side. When testing the service a terminator who is called by the subscriber is needed.

Note that the service only handles the case where the caller picks up the receiver; the service does not know of services other than POTS so making it handle other ways of creating calls, e.g., TWC calls, will be a solution to an interaction.

Terminating Call Screening (TCS)

The Terminating Call Screening service allows the subscriber to block incoming calls based on the identity of the caller, i.e. the number. A caller who is screened will get an announcement that the call is not allowed and the call will not be established.

The service definition is displayed in Figure 6.12.

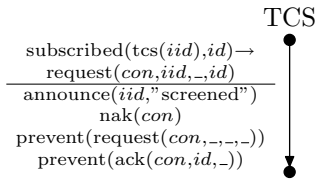

Figure 6.12: The TCS service.

The subscriber is associated with a collection of screened numbers, i.e. numbers that are not allowed to connect to the subscriber.

The service is armed when it is subscribed.

The service is activated when it is armed and a *request* from a screened number is observed by the subscriber. When activated and the originator of the call is

blocked from the subscriber, an announcement will be made to the caller and the connection will not be established.

The service operates on the terminating side. When testing the service an originator calling the subscriber is needed.

Three-Way Calling (TWC)

The Three-Way Calling service allows the subscriber to put an active connection on hold, establish a new connection and then merge the new connection with the connection on hold into one active connection. If the subscriber hangs up with a user on hold, a special alert will be started reminding that there exists an unterminated connecting.

The service definition is displayed in Figures 6.13, 6.14 and 6.15.

The service is armed when it is subscribed but not active and the subscriber is engaged in a call (has a voice path).

The service is activated when it is armed and a $flash$ event is observed. When activated the service puts the active connection on hold and allows the subscriber to establish a new connection which can be merged with the one on hold. TWC manages the initialisation of the new call and no POTS is created for that call. If the subscriber hangs up while a connection is on hold, an alert is started. If the hook is lifted when the alert is active, TWC will prevent POTS from initiating a new call.

The service requires the subscriber to be able to put a call on hold and it also requires the subscriber to be able to relay the special alert.

The service operates on either the terminating or the originating side. When testing the service actors to constitute the original call and a terminator to be called by the subscriber are needed.

Note that in our implementation TWC will take the role as a POTS for the establishment and maintenance of the connection to the third user. The call will make the call into a normal POTS instance at the earliest possible moment, i.e. when one of the other users disconnect.

The service will interact with services that require that calls are established with POTS.

IN Call Forwarding (INCF)

The IN Call Forwarding service forwards all incoming call.

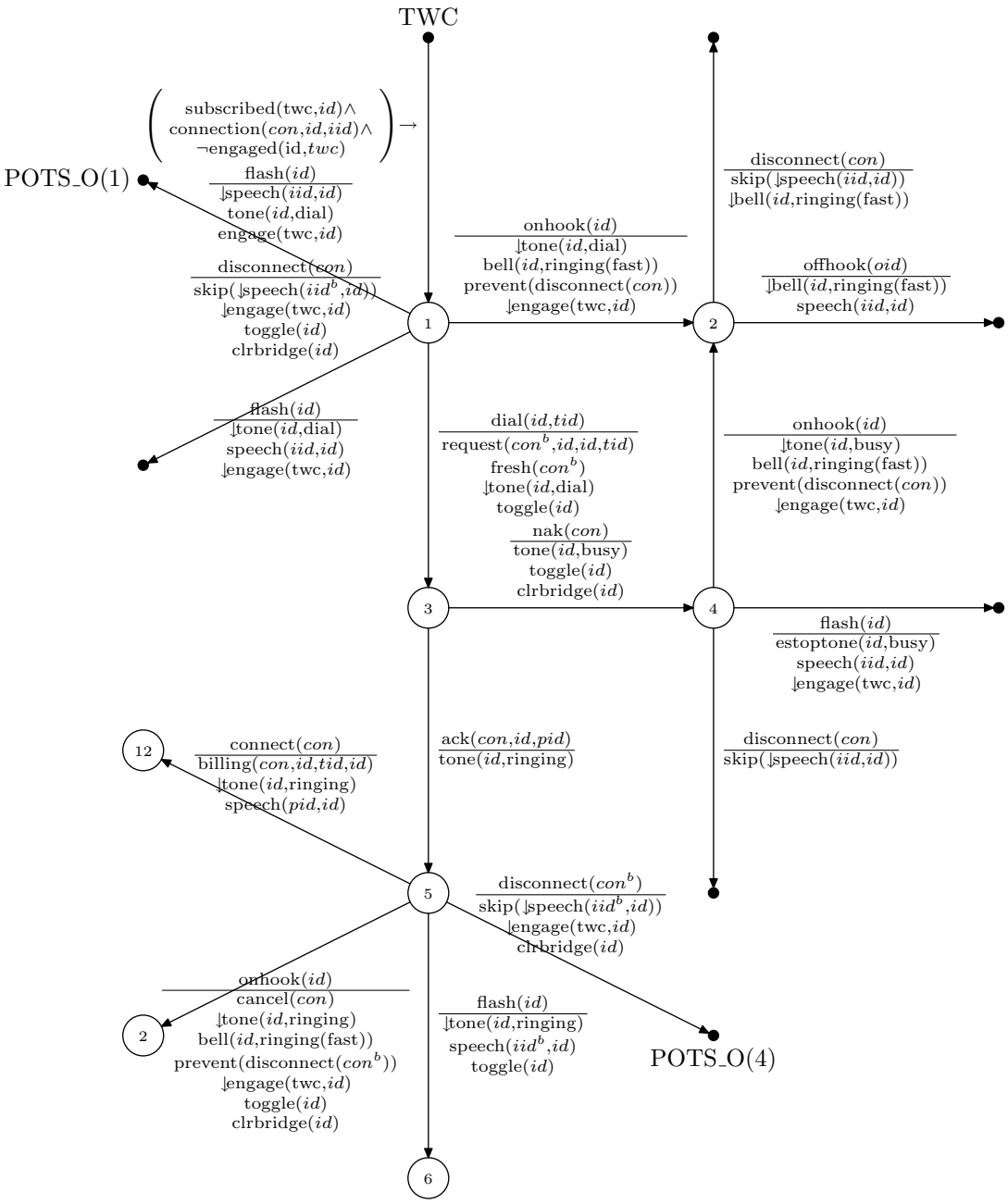

Figure 6.13: The TWC service.

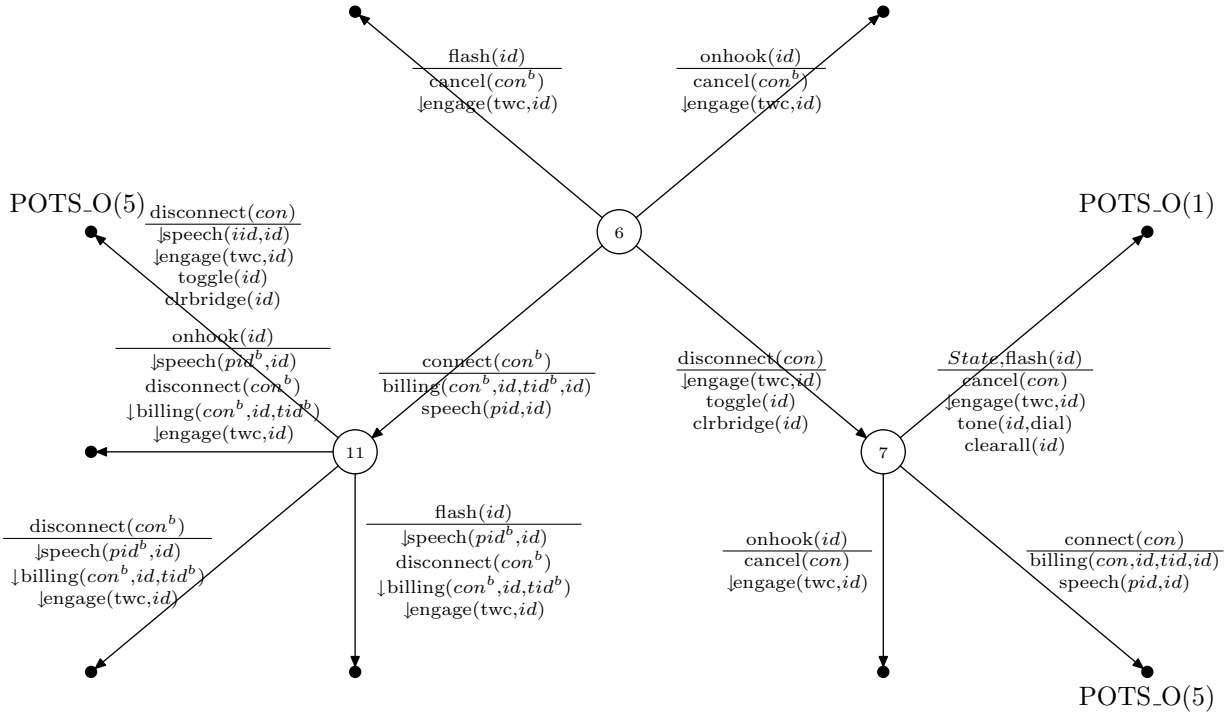

Figure 6.14: The TWC service (cont).

The definition of the service is shown in Figure 6.16.

The subscriber is associated with a number, *tid*, to which the service forwards calls when activated.

The service is armed when it is subscribed.

The service is activated when it is armed and a *request* event to the subscriber is observed. Upon activation it will extend the incoming connection with a leg to *tid* and prevent the standard POTS behaviour to nak busy terminators.

The service is a forwarding service and requires the CF feature, cf. 6.2.1, which handles the service behaviour after activation.

The service operates on the terminating side. When testing the service an originator, making a call to the busy subscriber, is needed as well as two terminators to whom the call is forwarded.

The service will when activated interact with other services trying to modify the connection behaviour at that point.

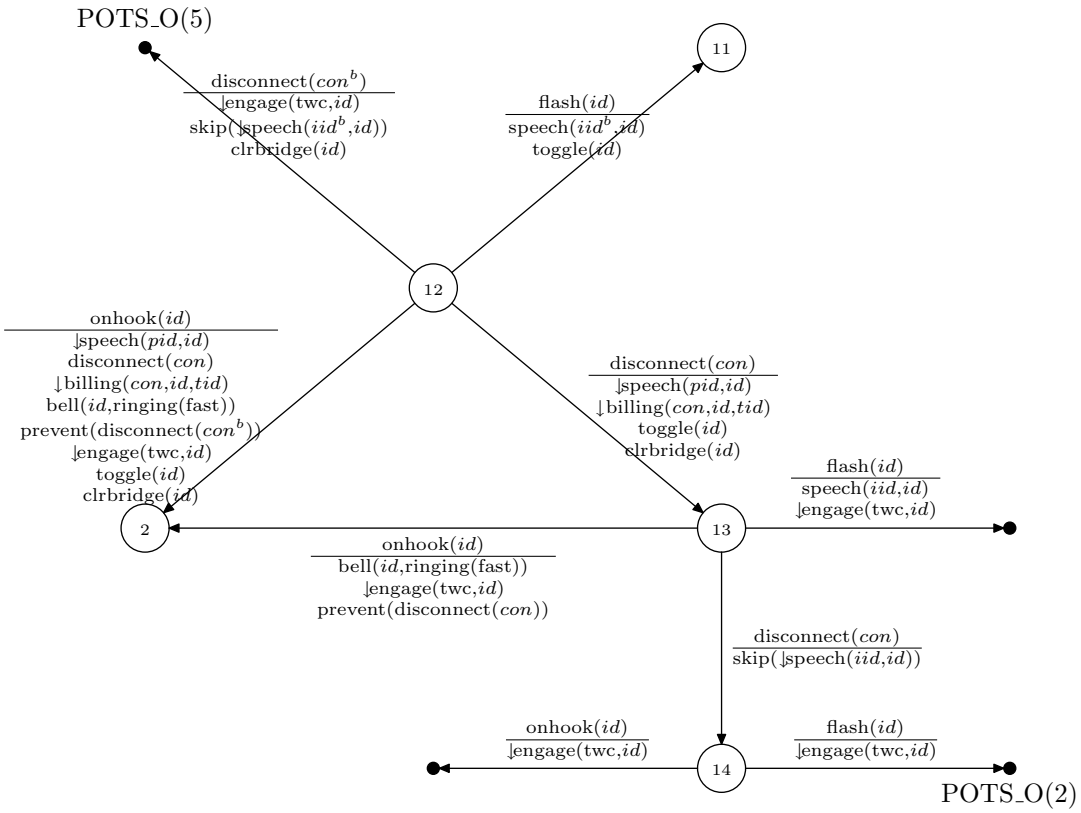

Figure 6.15: The TWC service (cont).

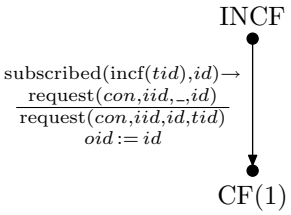

Figure 6.16: The INCF service.

Call Waiting (CW)

The Call Waiting service alerts the subscriber of an incoming connection when the subscriber is engaged a connection. The service then allows the acceptation of the incoming call by putting an already active connection on hold. The subscriber can then switch the active connection with the one on hold.

The definition of the service is shown in Figure 6.17.

The service is armed when it is subscribed but not active and the subscriber is engaged in a call (has a voice path).

The service is activated when it is armed and a *request* event is observed. When activated the service alerts the subscriber with a special tone and then enables the subscriber to put the active connection on hold to accept the incoming call and then alternate between the two calls using the flash button.

The service requires the subscriber to be able to put a call on hold and it also requires the subscriber to be able to relay the special alert.

The service operates on either the terminating or the originating side. When testing the service actors to constitute the original call and an originator who calls the subscriber are needed.

Our implementation creates a terminating POTS instance for the received call and the just manages both the connections until one user disconnects and CW can let POTS handle the rest. There are some situations where the second call will not become a POTS, and this happens when the first connection is terminated before the second one has been connected. When this happens, the CW will proceed to become an initiating POTS instance.

Charge Call (CC)

The Charge Call service allows the subscriber to charge a call to another account.

The definition of the service is shown in Figure 6.18.

The service associates accounts with PINs.

The service is armed when it is subscribed and the subscriber is about to make a call (has a dial tone)

The service is activated when it is armed and the charge call service is dialled. When activated the service will ask for an account, *account*, and an associated PIN, PIN. If a valid account and PIN is dialled the service will charge the account and prevent the normal POTS billing. If a bad account or PIN is dialled the subscriber will not be able to originate calls and will be assumed to hang up.

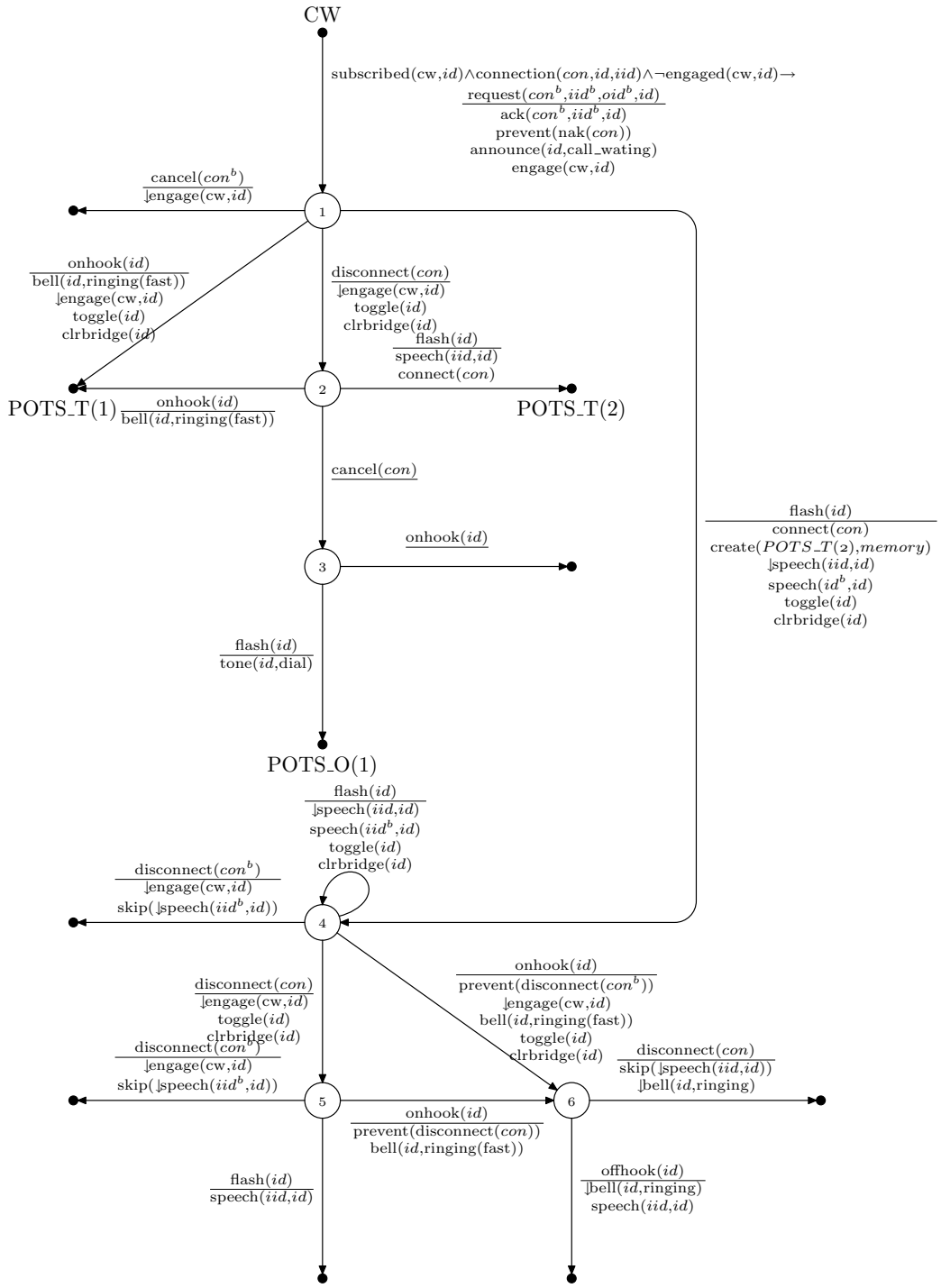

Figure 6.17: The CW service.

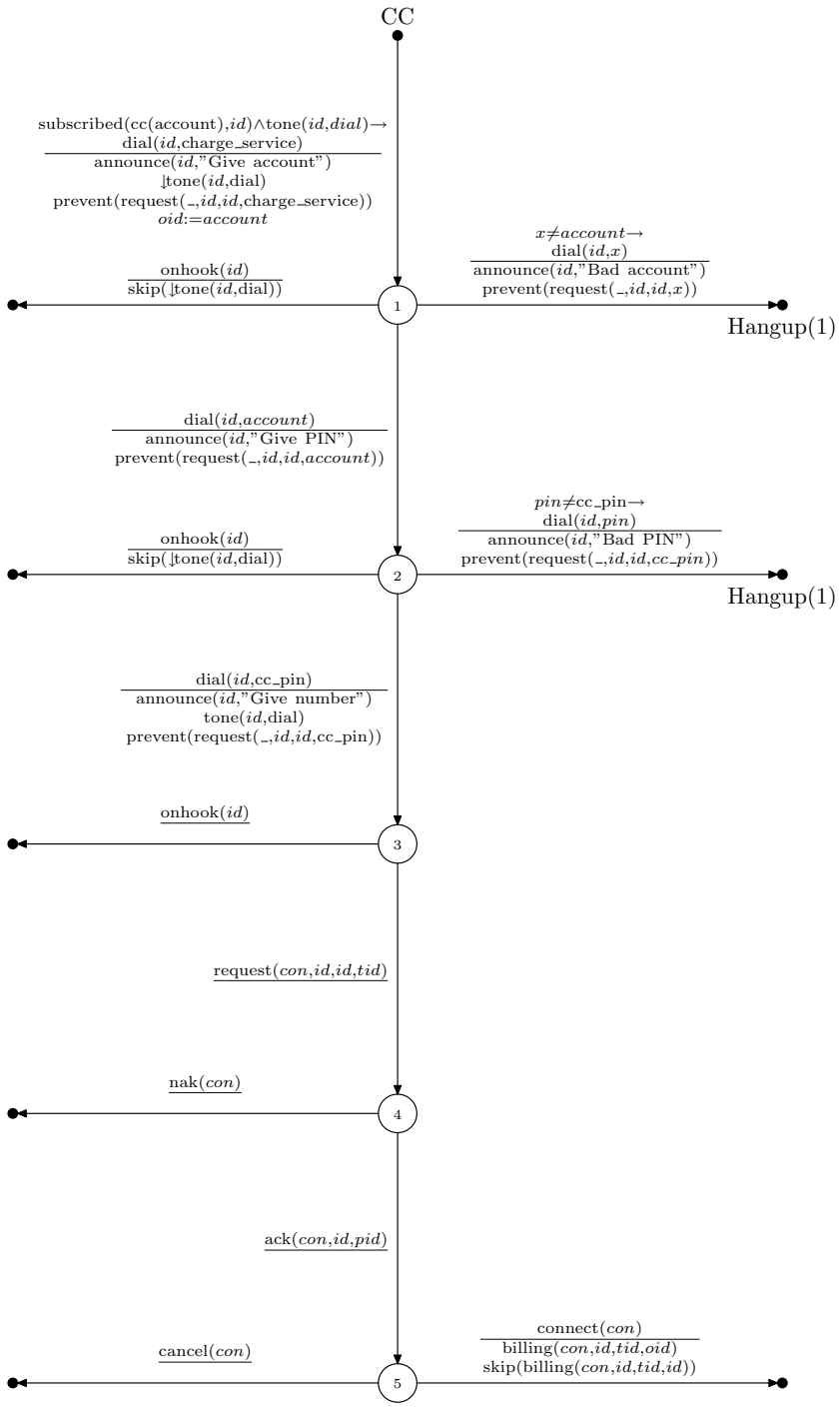

Figure 6.18: The CC service.

The normal POTS behaviour will be prevented while gathering account and PIN information.

The service require the HANGUP feature as this service is used when a bad account or PIN has been dialled. The service also requires an announcement functionality so that it can alert the subscriber. The service also requires a special number which can be dialled to activate the service.

The service operates on the originating side. When testing the service a terminator who is called by the subscriber is needed.

In our implementation the service works like Account Card Calling; the service number is called, the account is given and finally the PIN. If anything is not correct on the way, the call is terminated.

Return Call (RC)

The Return Call service allows the subscriber to recall the latest incoming call that was not received. If the call cannot be recalled directly it will alert the user when the call can be made.

The definition of the service is shown in Figure 6.19.

The service is armed when it is subscribed.

The service is activated when it is armed and a *ack* event of the subscriber is observed. Upon activation it will remember the originator of the incoming call and if the connection is not established it will allow the user to all the return call service to attempt a connection to the remembered number. If the connection can not be established it will announce this to the subscriber. If the subscriber then hangs up it will alert when the connection can be established.

The service requires that announcements can be made, that the special alert can be relayed and it also requires a special number which can be dialled to initiate the return call.

The service operates on the terminating side. When testing the service an originator who makes the activating call and is later called by the subscriber is needed.

Our implementation has two phases, the triggering phase where the incoming call is remembered and the recall phase where the actual recall is attempted. The recall phase can loop back to the triggering phase if another incoming call is made before the recall has been initiated of if the user aborts a recall.

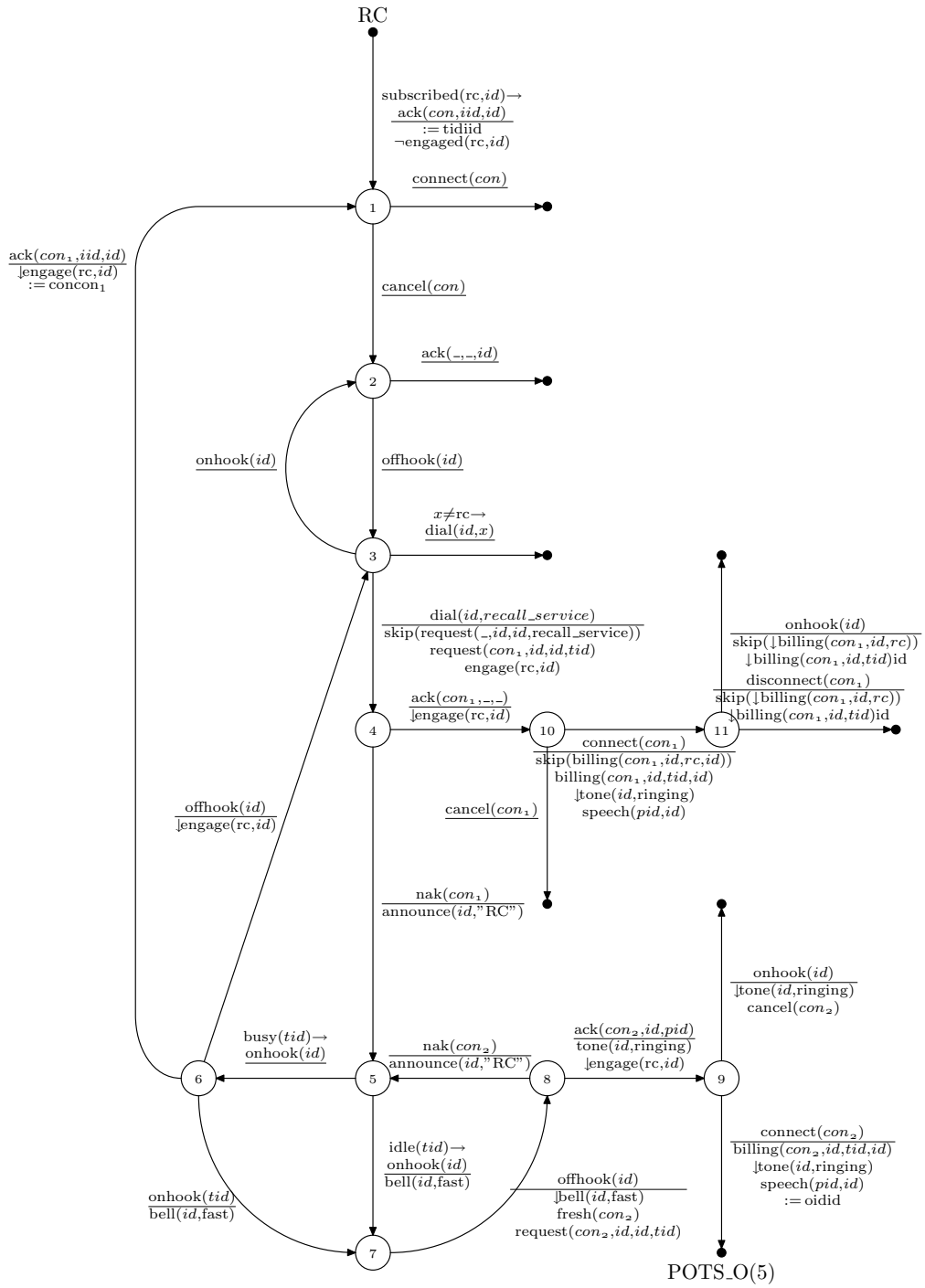

Figure 6.19: The RC service.

Cellular Phone (CELL)

The Cellular Phone service adds air-time to the charge of a (connection) leg.

The definition of the service is shown in Figure 6.20.

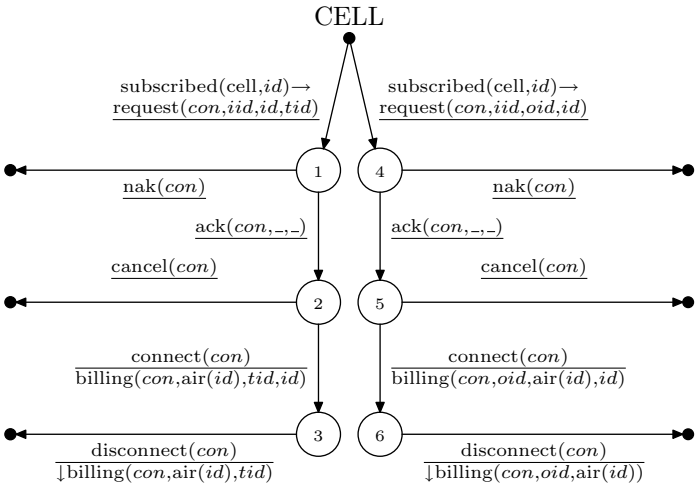

Figure 6.20: The CELL service.

The service is armed when it is subscribed.

The service is activated when a *request* event which has the subscriber as originator or terminator is observed. Extra billing will be added for the connection if it is established.

The service requires that extra billing can be started.

The service operates on both the terminating and the originating side. When testing the service actors to constitute the call are needed.

6.2.1 Features

Here we describe two process definitions that will be used in several of the preceding services.

Call Forward (CF)

In itself Call Forwarding, Figure 6.21, is not a service, it has no start of it own. It has to be started by a service like INCF or CFBL. When ringing it will handle the forwarding between the subscriber and the user she is forwarded to.

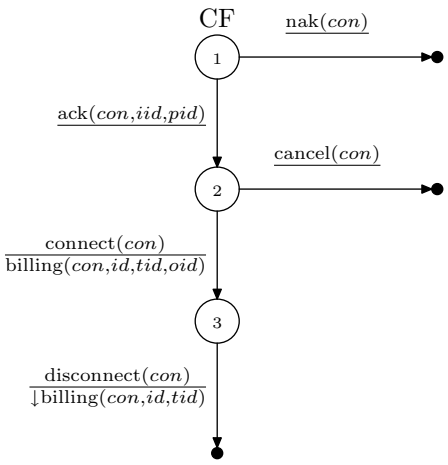

Figure 6.21: The CF feature.

Since the control location CF(1) is shared by several services, we have chosen to present it separately in Figure 6.21. The purpose of the transitions in the diagram of CF is to carry out actions to the effect that user *oid* is billed for the connection leg between *oid* and *tid*. The automaton observes the establishment of connection *con*. When the final *connect* message is generated, it triggers a *billing* action, which starts this billing. Correspondingly, a *disconnect* message triggers a $\downarrow$*billing* action, which stops billing.

Hangup (HANGUP)

The HANGUP feature, Figure 6.22, waits until the user has hanged up. Until that has happened it blocks all attempts to make calls.

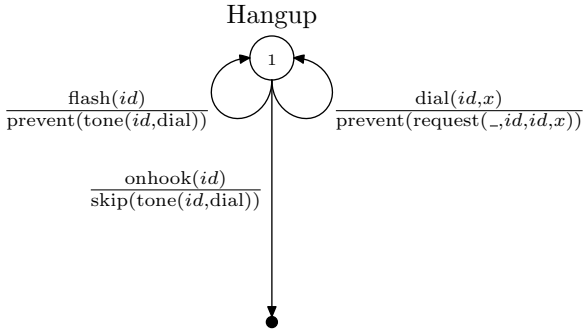

Figure 6.22: The HANGUP feature.

6.3 INTERACTION DETECTION RESULTS

After (re)modelling the contest services in our framework we searched for interactions between all pairs of services. We repeatedly analysed the error trails to resolve interactions and then search again for new interactions. This cycle of detection and resolution was exercised a few times and in order to measure the strength of our method. A comparison to the interactions reported by the contest committee [GBG+00], here used as a benchmark, is presented in Table 6.1.

Table 6.1: Interactions in the case study.

Service	CELL	RC	CC	CW	INCF	TWC	TCS	INTL	INFR	INFB	CND	CFBL
CFBL	2^{1}	1^{1}	1	1^{1}	3•	4	2	0^{1}	3•	2	1	$0^{1\dagger}$
CND	0	1	1	1	2	1	1	0	2	0	0	
INFB	2	1	1	1	2	1	1	0	2	0		
INFR	2	2	2	1•	3•	3	2	0	2•			
INTL	0	1^{1}	1○	0	0^{1}	1	0	0				
TCS	0	1	1	1	2	1	0					
TWC	4	1	2	9○	3	1						
INCF	2	2	1	1•	1•							
CW	3	0	1	0								
CC	0	1	0									
RC	0	0										
CELL	0											

The following notation is used in the table:

A^{B} denotes that the contest committee described A interactions in the contest model. The B denotes that the committee also identified B real-system interactions (not present in the contest model).

• denotes interactions we have detected and resolved.

○ denotes that an interaction is reported in the benchmark which is not detected by our method.

† denotes an interaction that was detected which we have not resolved.

At a glance it looks as if our framework does a real lousy job since it detects so few interactions. The rationale for this is that many of the benchmark interactions are artifacts of how the system is modelled. The benchmark's artifact interactions are spurious and avoided in our model. Out of the 105 interactions identified and described in the benchmark (both interactions in the contest model and in real systems), we have found that *our model of the system avoids around 80% of the interactions.*

To illustrate this we will look at the first column of interactions, all interactions where Cellular Phone is involved.

Cellular Phone **interactions**

The benchmark interactions reported for the Cellular Phone (CELL) service will now be presented categorised by the other service(s) involved in the interaction.

CFBL, INFR and INCF Interactions are caused by the way CELL is defined. The contest definition of the CELL service assumes that connections have only one connection leg and when the forwarding services create a connection of two legs this assumption is broken, a situation for which the CELL service is not defined and due to this it will fail to start the air-time billing.

These interactions are not present in our model since all services are designed to work with the base system concept of connections of more than one connection leg.

The real-system interaction not present in the contest with CFBL occurs when CELL does not add its air-time billing because it is not the caller or callee of the connection[1].

This interaction is not present in our model since our definition of the CELL service activates dynamicly and our activation condition properly identifies the cases where forwarding occurs. Further discussions on this is found below, in the description of the interactions with TWC and CW.

INFB Both interactions are caused by the way in which the services are defined. CELL is defined to take over POTS behaviour and therefore tries to start (POTS's) normal billing as well as its air-time billing. INFB is also designed to take over POTS behaviour and will substitute POTS's billing for a reversed one. The interactions occur when CELL tries to bill normally and INFB tries to bill reversed.

These interactions are not present in our model since services never take over the behaviour of POTS[2] and hence the billing modifications introduced by the services will not conflict.

TWC The first pair of interactions are caused by the service composition described and used by the contest. CELL is defined to start from a normal POTS call, i.e. all the way from when the receiver is lifted off the hook. This is the

[1] Connections described by the contest can be extended to at most two legs by forwarding, so the interaction occurs when CELL is subscribed to by the phone *in the middle*, i.e. the forwarding phone.

[2] Unless they really have too and in such cases they are responsible that their own version of POTS *works*.

only way CELL assumes that connections can be initiated. The call TWC initiates after the $flash$ will not be recognised and hence CELL will not start its billing.

These interactions are not present in our model since services are activated when their activation conditions are met. CELL is activated when connection requests are made, which all connections will have to be initiated with.

The second pair of interactions occur when the billing (not started above) is not correctly terminated.

CW The first two interactions are similar to the interactions with INFB. CW allows incoming calls to be accepted with a $flash$ and this is not the way CELL assumes phone calls to be accepted, CELL expects an *offhook*.

These interactions are not present in our model since CELL assumes that connections are accepted with a *connect*.

The third interaction is similar to the first two; the billing added by CELL is not correctly terminated when the second call ,accepted using CW, is not terminated in the normal POTS way.

This interaction in not present in our model since CELL assumes that connections are ended with *disconnect*.

A note on the benchmark interactions is that they are incomplete. A clear indication of this is that the set of interactions reported for the different forwarding services are not the same. The forwarding loop was only identified for CFBL, not for INFR and INCF. We detected loops for all forwarding services, though this is not indicated in Table 6.1.

Resolved interactions (•)

To validate that our tool worked and allowed easy resolution of interactions we resolved forwarding interactions with simple priority assignment. There are three forwarding services described in the contest: INCF, INFR and CFBL.

The interactions among the forwarding services were resolved by assigning a priority order among these services. INCF was given priority over the other two services and INFR was given priority over CFBL and with this priority assignment the interactions did not occur.

The interactions between the forwarding services INCF and INFR, and CW were resolved by assigning the forwarding services priority over CW, thereby preventing CW from accepting the incoming call.

The interactions in a scenario where two instances of the same forwarding service, e.g., CFBL and CFBL forwards to two different phones will not be generated by our scenario generator.

The situation between CW and CFBL, where we do not detect any interactions can be explained with that the contest model interaction is created by how CFBL is activated by the contest model, and the the real-system interaction between the two services is created by how CW is activated in the model. Both these interactions are avoided by our dynamic activation of services.

Undetected interactions (∘)

INTL **and** CC This interaction is not detected since we have separate PINs for all services. This is indicates that there are scenarios which will not be generated[3].

TWC **and** CW There is one interaction which is not detected between these two services. The benchmark describes interactions in situations where the subscriber to both services, B, hangs up on someone, D, with someone on hold, C, and with an incoming call, A.

Detected unresolved interactions (†)

CF The real-system interaction indicated for CFBL is the forwarding loop. This interaction is not present in the contest model since connections can have at most two connection legs.

The interaction is detected in our model since the CF feature creates a loop of intermediate system states. The only way to resolve the solution is to extend either the forwarding feature or the system but this was outside the scope of our case study.

Note that we detect this interaction for all services using the CF feature and that detecting this interaction only needs one instance of the forward forward feature, forwarded to itself.

6.3.1 Observations

During the case study the framework showed that it together with simple rules of mind can make development of features easier.

Like Capellmann et al.[CCP+97] we also belive that the greatest time consumption when using formal methods lies in the modelling and not in the analysis. The

[3]Knowing that this interaction existed it was easy to give the services the same PIN and rerun the exploration which then detected the interaction.

analysis is done in a matter of days once the model of the system has been properly captured and checked, a task which takes much longer.

Rules of thumb during feature design

Some of the features in the contest were flawed and needed improvements or modifications to work correctly. The rules below were both easy to define and follow.

1. Phones can either do an *offhook* event or *onhook* event in all quiescent system states and these states can be traced to specific control locations in the services. Control locations where this property is not fulfilled are likely to be erroneous since users assume they always can end whatever they do by hanging up. If a service need attention after the hook has been hanged up, users expect that the service will make this known.
2. Phones will not be able to generate spontaneous events in intermediate system states and hence transitions triggering on this can be removed.
3. Connection events come in sets that should appear together. Since it can not be certified that a connection will get an *ack* or a *nak* in an intermediate system state, all control locations triggering on *ack* event should also trigger on a *nak*.
4. When a feature jumping into another feature it must have a global system view which is consistent with the view of the feature entered, e.g., a service who assumes that a phone is off the hook should not jump to a control location where the same phone is on the hook without responding the correct *onhook*.
5. All events, variables and values used must be defined or else interactions are likely to occur.

Rules like these aid can be used to help the developer of a service avoid common mistakes in service design.

During the remodelling of the services of the second contest, this kind of checks were invaluable since these services contained several flaws. We detected both situations where the features were obviously syntactically bad and situations where the logic of the features were compromised.

Flexibility

The graphical notation gives the feature designer an easy way to sketch features by themselves, i.e. as stand alone pieces of functionality. This property seems to

reduce the number of mistakes made when designing features, though we can not show this to be the case. What we know is that we easily identified several design flaws among the services of the second interaction detection contest.

The graphical notation and the fact that components are designed stand alone in the framework allows changes to be made in a flexible way. New transitions can easily be added to the components and new detail can be incorporated into the model seemingly effortless—or so we believe.

Requirements

Though requirements seem to be a great idea, they often seem to be one step behind in detecting interactions, i.e. when they report an interaction it can mostly be detected earlier using the other interaction detection methods we use. Nevertheless, we found requirements an excellent aid when testing if features worked correctly with respect to the base system. As an example we identified a case where the definition of a service given in the contest did not terminate its billing correctly.

A problem we also identified with requirements is that they require strategies for initial tuning to the system model, and further strategies for keeping them in tune with the system if it is changed.

Chapter 7

Conclusions

In this thesis I have presented a framework for modelling telecommunication systems. The framework we have designed is based on general concepts that should allow it to be used when modelling any system with communicating processes were communication can be accomplished by passing between the processes. We believe that this goal has been met as we have been able to describe a model of a telephony system.

We have described and reasoned about how interactions manifest themselves in the framework and the way in which we describe them is not domain specific; interaction caused by inconsistent or inapplicable events can be used in any system modelled using the framework. The use of requirements can be used to detect logical interactions.

A model of telecommunication systems has been described. A primary goal in the design of the model has been to make it intuitive by deriving its design from observations of common telecommunication concepts. A presentation of the design process has been given and it resulted in a model which consists of what we consider useful functionality. A second goal in the design process has been to allow services to be developed as stand alone pieces of functionality. This has been met and has proven vital during the case study we did.

Observations indicate that extra methods like scenario selection and symmetry elimination can be used with the framework and model to make exhaustive state space exploration feasible.

In a case study, we show how the telephony system and services of the first feature interaction detection contest [GBG+00] can be modelled in the framework. Results from the analysis show that the model we use avoids around 80% of the spurious interactions presented in a benchmark for the contest. We have found that our results are based on the fact that we did an analysis of the functionality needed in the base system and implemented this so the services could be designed to work with concepts like connection legs from the start. The case study also showed that

our framework allowed us to identify flaws in the original feature definitions, both syntactic and semantic ones.

We find that this to be an important contribution when modelling a system. We believe that our graphical notation of features adds value to the framework since it makes it easy to visualise the definition of features which makes them easier to understand and work with.

Another goal of the framework was to make it flexible in the way changes, like change of granularity, can be made to the modelled system. During the work with the case study we found it easy to add new functionality to low layer components and we take this fact as confirmation of the framework's flexibility.

We strongly believe that this framework, based on our encouraging results using it, is a contribution to the area of feature interactions detection in telecommunication systems.

Bibliography

[AGG+98] A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne, "SCF3^{TM}/Sculptor with Chisel: Requirements Engineering for Communications Services", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 45–63, IOS Press, 1998.

[A-K91] H. Aït-Kaci, "Warren's Abstract Machine—A Tutorial Reconstruction" The MIT Press, 1991.

[AVW+96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. "Concurrent Programming in Erlang", Prentice Hall, 1996.

[BrAt94] K.H Braithwaite, and J.M Atlee, "Towards Automated Detection of Feature Interactions", *Feature Interactions in Telecommunications and Software Systems*, L.G. Bouma and H. Velthuijsen, eds., pp. 36–59, ISO Press, 1994.

[BAE+98] R. Buhr, M. Amyot, D. Elammari, T. Quesnel, and S. Gray, "Feature-Interaction Visualization and Resolution in an Agent Environment", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 135–149, IOS Press, 1998.

[BEG+98] R. Buhr, M. Elammari, T. Gray, and S.Mankovski, "Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example", *Proc. HICSS'98)*, IEEE, 1998.

[Cal98] M. Calder, "What Use are Formal Design and Analysis Methods to Telecommunications Systems", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 23–31, IOS Press, 1998.

[CGL+93] E.J. Cameron, N. Griffeth, Y.-J. Lin, M.E. Nilson, W.K. Schnure, and H. Velthuijsen, "A Feature-Interaction Benchmark for IN and

Beyond", *IEEE Communications Magazine*, vol. 31, no. 3, pp. 64–69, Mar. 1993.

[CGL+94] E.J. Cameron, N. Griffeth, Y.-J. Lin, M.E. Nilson, W.K. Schnure, and H. Velthuijsen, "A Feature-Interaction Benchmark for IN and Beyond", *Feature Interactions in Telecommunications and Software Systems*, L.G. Bouma and H. Velthuijsen, eds., pp. 1–23, ISO Press, 1994.

[CaVe93] E.J. Cameron, and H. Velthuijsen. Feature Interactions in Telecommunications Systems. *IEEE, Communications Magazine*, vol. 31, no. 8, pp. 18–23, Aug. 1993.

[CCP+97] C. Capellmann, P. Combes, J. Pettersson, B. Renard, and J.L. Ruiz, "Consistent Interaction Detection - A Comprehensive Approach Integrated with Service Creation", *Interactions in Telecommunications Systems IV*, P. Dini, R. Boutaba, and L. Logrippo, eds., pp. 183–197, IOS Press, 1997.

[ChLu95] T-Y. Cheung, and Y. Lu, "Detecting and Resolving the Interactions Between Telephone Features Terminating Call Screening and Call Forwarding by Colored Petri-nets", pp. 2245–2250, IEEE, 1995.

[Gib97] J.P. Gibson, "Feature Requirements Models: Understanding Interactions", *Interactions in Telecommunications Systems IV*, P. Dini, R. Boutaba, and L. Logrippo, eds., pp. 46–60, IOS Press, 1997.

[GBG+00] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta, "A feature interaction detection benchmark for the first feature interaction detection contest", *Computer Networks*, 32(4), pp. 389–418, 2000.

[GBG+98] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta, Feature Interaction Detection Contest, *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 237–259, IOS Press, 1998.

[GBM+99] C. Gottbrath, J. Bailin, C. Meakin, T. Thompson, and J.J. Charfman, "The Effects of Moore's Law and Slacking on Large Computations", Steward Observatory, University of Arizona, Dec. 1999.

[Hal98] R.J. Hall, "Feature Combination and Interaction Detection via Foreground/Background Models", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 232–246, IOS Press, 1998.

[Kec98] D.O. Keck, "A Tool for the Identification of Interaction-Prone Call Scenarios", *Feature Interactions in Telecommunications and Software*

Systems V, K. Kimbler and L.G. Bouma, eds., pp. 276–290, IOS Press, 1998.

[Kho97] A. Khoumsi, "Detection and Resolution of Interactions Between Services of Telephone Networks", *Interactions in Telecommunications Systems IV*, P. Dini, R. Boutaba, and L. Logrippo, eds., pp. 78–92, IOS Press, 1997.

[Kim97] K. Kimbler, "Adressing the interaction problem at the enterprise level", *Interactions in Telecommunications Systems IV*, P. Dini, R. Boutaba, and L. Logrippo, eds., pp. 13–22, IOS Press, 1997.

[KPR97] C. Klein, C. Prehofer, and B. Rumpe, "Feature Specification and Refinement with State Transaction Diagrams" *Interactions in Telecommunications Systems IV*, P. Dini, R. Boutaba, and L. Logrippo, eds., pp. 284-297, IOS Press, 1997.

[KMM+00] M. Kolberg, E.H. Magill, D. Marples, and S. Reiff, "Second Feature Interaction Detection Contest", *Feature Interactions in Telecommunications and Software Systems VI*, pp. 293–310, IOS Press, 2000.

[LiCa98] L. Lima, A. Cavalli, "Application of embedded testing methods to service validation", *Proc. 2nd IEEE International Conference on Formal Engineering Methods*, pp. 212–221, 1998.

[LLG98] F.L. Lin, H. Liu, and A. Ghosh, "A Methodology for Feature Interaction Detection in the AIN 0.1 Framework", *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 797–817, Oct. 1998.

[MaPn95] Z. Manna, and A. Pnueli, "Temporal Verification of Reactive Systems", Springer Verlag, 1995.

[MaMa98] D. Marples, and E.H. Magill, "The use of Rollback to prevent incorrect operation of Features in Intelligent Network Based Systems", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 115–134, IOS Press, 1998.

[NKK97] M. Nakamura, Y. Kakuda, and T. Kikuno, "Petri-Net Based Detection Method for Non-Deterministic Feature Interactions and its Experimental Evaluation", *Interactions in Telecommunications Systems IV*, P. Dini, R. Boutaba, and L. Logrippo, eds., pp. 138–152, IOS Press, 1997.

[NKK98] M. Nakamura, Y. Kakuda, and T. Kikuno, "Feature Interaction Detection Using Permutation Symmetry", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., ⟪pp. 187–201⟫, IOS Press, 1998.

[NyJo96] J. Nyström, and B. Jonsson, "A Formalization of Service Independent Building Blocks", *Proc. International Workshop on Advanced Intelligen Networks*, T. Margaria, ed., pp. 1–14, Mar. 1996.

[OtHa94] T. Ohta, and Y. Harada, ""Classification, Detection and Resolution of Service Interactions in Telecommunication Services", *Feature Interactions in Telecommunications and Software Systems*, L.G. Bouma and H. Velthuijsen, eds., pp. 60–72, ISO Press, 1994.

[PlRy98] M. Plath, and M. Ryan, "Plug-and-play features", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 150–151, IOS Press, 1998.

[Q1200] ITU-T, "Recommendation Q.1200. XX", *General recommendations on telephone switching and signaling, Intelligent Network*, Mar. 1993.

[Q1213] ITU-T, "Recommendation Q.1213. Global functional plane for Intelligent Network CS-1", *General recommendations on telephone switching and signaling, Intelligent Network*, Mar. 1993.

[Q1214] ITU-T, "Recommendation Q.1214. Distributed functional plane for Intelligent Network CS-1", *General recommendations on telephone switching and signaling, Intelligent Network*, Mar. 1993.

[Rei00] S. Reiff, "Notes on Call Configurations with Features", *Proc. Workshop on Language Constructs for Describing Features*, S. Gilmore and M. Ryan, eds., pp. 71–77, 2000.

[WFK+93] Y. Wakahara, M. Fujioka, H. Kikuta, H. Yagi, and S.-I. Sakai, "A Method for Detecting Service Interactions", *IEEE Communications Magazine*, vol. 31, no. 8, pp. 32–37, Aug. 1993.

[Zav93] P. Zave, "Feature Interactions and Formal Specifications in Telecommunications", *IEEE Computer*, vol. 26, no. 8, pp. 20–29, Aug. 1993.

[Zav95] P. Zave, "Secrets of call forwarding: A specification case study", *Proc. FORTE '95*, pp. 153–168, Chapman & Hall, 1995.

[Zav98] P. Zave, "Architectural Solutions to Feature-Interaction Problems in Telecommunications", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 10–22, IOS Press, 1998.

[ZWR+] I. Zibman, C. Woolf, P. O'Reilly, L. Strickland, D. Willis, and J. Visser, "An Architectureal Approach to Minimizing Feature Interactions in Telecommunications", IEEE/ACM Transactions on Networking, vol. 4, no. 4, pp. 582–596, Aug. 1996.

[Z-M98] R. Zygan-Maus, "Feature Interaction Management for Public Switching Networks", *Feature Interactions in Telecommunications and Software Systems V*, K. Kimbler and L.G. Bouma, eds., pp. 32–44, IOS Press, 1998.

Licentiate theses from the Department of Information Technology

2000-001 Katarina Boman: *Low-Angle Estimation: Models, Methods and Bounds*

2000-002 Susanne Remle: *Modeling and Parameter Estimation of the Diffusion Equation*

2000-003 Fredrik Larsson: *Efficient Implementation of Model-Checkers for Networks of Timed Automata*

2000-004 Anders Wall: *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*

2000-005 Fredrik Edelvik: *Finite Volume Solver for the Maxwell Equations in Time Domain*

2000-006 Gustaf Naeser: *A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems*

UPPSALA
UNIVERSITY

Department of Information Technology, Uppsala University, Sweden