

# Language Constructs for Defining Features for Reactive Systems

Malte Plath

A thesis submitted to the  
School of Computer Science  
of the University of Birmingham  
for the degree of Doctor of Philosophy

School of Computer Science  
The University of Birmingham  
Birmingham B15 2TT  
United Kingdom

October 2000



---

## Abstract

In an age of ever bigger and more complex software and hardware systems, extending an existing system with new functionality has become very important. A *feature* is a unit of functionality that can be added to an existing system, and *feature-oriented specification* tackles the problem of how to add new functionality to a system in a systematic way. The telecommunications industry, for example, has incrementally upgraded extensive networks with new services. Today services, or ‘features’, such as Call Waiting or Voice Mail are available to most users. Some combinations of services, however, produce unexpected results. This phenomenon has become known as *feature interaction*.

Features and feature interactions can be found not only in telephone networks but can occur in virtually any system that is enhanced with new or optional functionality. Adding functionality always brings with it the possibility that the new functionality interferes with other operations of the system.

These observations raise three central questions:

- how to specify features for a system,
- how to add features to the system, and
- how to detect interactions among features.

The method to answer to these questions proposed here is to extend specification languages with *feature constructs*. Each feature construct is complemented by a notion of *feature integration*, that is, a method for adding a feature definition to the specification of a system. Feature integration gives semantics to a feature construct, which in turn can be used to prove general properties of features. This thesis develops feature constructs for three specification languages, SMV, PROMELA and CSP, and demonstrates their use in examples and case studies. For SMV and CSP, formal semantics of the feature constructs are developed, too.

For detecting feature interactions, the potential of model checking is explored. Model checking offers an automated method to prove properties of the system before and after feature integration. In theory, this could be used to test for any combination of features whether the features work correctly. In practice, however, it is often not clear what ‘correct’ operation means, and moreover, for large systems model checking becomes intractable.



# Contents

<b>Acknowledgements</b>	<b>vi</b>
<b>1 Features and Feature Interaction</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Features . . . . .	1
1.3 Feature interaction . . . . .	3
1.4 Reactive systems . . . . .	3
1.5 Existing approaches . . . . .	5
1.5.1 Offline detection . . . . .	6
1.5.2 Online detection . . . . .	9
1.6 Verification and validation . . . . .	9
1.6.1 Specification languages . . . . .	10
1.6.2 Temporal logics . . . . .	10
1.6.3 Model checking . . . . .	12
1.6.4 Simulation and testing . . . . .	13
1.7 Specifying features . . . . .	14
1.7.1 Feature constructs . . . . .	14
1.7.2 Feature integration . . . . .	15
1.7.3 Model checking approach to feature interactions . . . . .	15
1.7.4 Related work . . . . .	15
1.8 The rest of this thesis . . . . .	17
1.8.1 Outline . . . . .	18
<b>2 Features for SMV</b>	<b>20</b>
2.1 Introduction . . . . .	20
2.2 The SMV model checker . . . . .	21
2.2.1 The SMV language . . . . .	21
2.3 A feature construct for SMV . . . . .	23
2.3.1 Syntax of the feature construct . . . . .	24
2.3.2 Integrating a feature . . . . .	25
2.3.3 Integration of multiple features . . . . .	27

## CONTENTS

---

2.3.4	Detecting feature interaction . . . . .	27
2.4	Case studies . . . . .	28
2.4.1	A lift system with five features . . . . .	28
2.4.2	A telephone system with eight features . . . . .	40
2.5	The semantics of the SMV feature construct . . . . .	58
2.5.1	The syntax of SMV revisited . . . . .	58
2.5.2	The semantics of SMV . . . . .	58
2.5.3	Semantics of the feature construct for SMV . . . . .	68
2.5.4	Properties of the feature construct for SMV . . . . .	72
2.6	Conclusions . . . . .	76
<b>3</b>	<b>Features for Promela/Spin</b>	<b>77</b>
3.1	Motivation . . . . .	77
3.2	The model checker SPIN . . . . .	78
3.3	The language PROMELA . . . . .	78
3.4	The feature construct for PROMELA . . . . .	80
3.4.1	Syntax of the feature construct . . . . .	80
3.4.2	Restrictions and practical considerations . . . . .	87
3.4.3	Conditionals and choices . . . . .	90
3.4.4	Atomic and deterministic sequences . . . . .	90
3.4.5	Miscellaneous issues . . . . .	91
3.5	Examples of features for PROMELA programs . . . . .	91
3.5.1	The lift system . . . . .	91
3.5.2	The telephone system . . . . .	95
3.6	Conclusions . . . . .	99
<b>4</b>	<b>Features for CSP</b>	<b>100</b>
4.1	Motivation . . . . .	100
4.2	The process algebra CSP . . . . .	101
4.2.1	The syntax of CSP . . . . .	101
4.3	The semantics of CSP . . . . .	104
4.3.1	Trace semantics of CSP . . . . .	105
4.3.2	Operational semantics of CSP . . . . .	107
4.4	A feature construct for CSP . . . . .	109
4.4.1	Ingredients of the feature construct . . . . .	109
4.4.2	Semantics of the feature construct . . . . .	111
4.4.3	The semantics of feature integration . . . . .	118
4.4.4	Miscellaneous issues . . . . .	119
4.5	Feature interaction contest . . . . .	120
4.6	The contest model . . . . .	120
4.6.1	Observations . . . . .	123

4.6.2	Static analysis . . . . .	125
4.7	Modelling POTS in CSP . . . . .	127
4.7.1	A network of basic call processes . . . . .	127
4.7.2	A specialised feature construct . . . . .	127
4.7.3	Feature integration . . . . .	131
4.7.4	Detecting interactions . . . . .	131
4.7.5	Limitations . . . . .	133
4.8	Feature interactions . . . . .	133
4.9	Conclusions . . . . .	135
<b>5</b>	<b>Conclusions</b>	<b>137</b>
5.1	Specify features with feature constructs . . . . .	137
5.2	The usefulness of feature constructs . . . . .	138
5.3	Benefits and limits of model checking . . . . .	138
5.4	Directions for further research . . . . .	140
<b>A</b>	<b>List of abbreviations</b>	<b>142</b>
	<b>Bibliography</b>	<b>144</b>

# Acknowledgements

## or: By way of saying Goodbye...

It is good to look back after more than three and a half years, and to realise how far I have come. The story of my PhD studies and my stay in Birmingham could be seen as a story of experimentation. Not only did my research involve quite a bit of that, the whole project of moving from Germany to Birmingham was an experiment. Only that in this one I was the experimenter and the subject, and this experiment did not involve so many telephones. Initial results after about two years were not promising, but I decided to stick it out. And to my surprise, it got a lot better. I leave Birmingham with some hesitation: I have made good friends here, and I have discovered some charming sides to life in Birmingham. There are too many people I would like to thank for making my time here a memorable one; I hope they know who they are.

Some people who have had a direct influence on my research, however, I will mention: my supervisor Mark Ryan for starting the whole project and for discussions, criticism and support; my contacts at BT Labs (Martlesham Heath): Ronnie Masson, Richard Kett and Simon Tsang, and their colleagues for giving me insights into the “real world” of telecommunications and fruitful discussions; Valeria de Paiva, Marta Kwiatkowska and Jeremy Wyatt who gave me valuable feedback as members of my Thesis Group; Muffy Calder and Alice Miller for letting me use and abuse their POTS model; many members of the ESPRIT working groups FIREworks and ASPIRE; my fellow research students for distractions, and for some useful advice; . . . and finally my parents and my grandmother for their moral support.

# Chapter 1

## Features and Feature Interaction

### 1.1 Introduction

This thesis is concerned with the specification of features for reactive systems, and with frameworks for feature-oriented specification, especially language constructs for defining features. In this chapter, I will explain the notions of “feature” and “feature-oriented specification”, and give an overview of the formal methods that can be used for specifying and verifying systems and their features.

### 1.2 Features

It is common practice in software development to extend the lifespan of products by enhancing and upgrading them with new functionality, or “features”. This is seen as a cheaper, quicker and easier solution than redeveloping the system from scratch. While the concept of features is applicable in many domains, it has mainly come out of the telecommunications industry where it is impossible to change the complete infrastructure at once, not least because parts of the system reside with millions of individual customers: telephones and private switches for office networks. It is therefore of paramount importance to “evolve” the system in such a way that compatibility with customers’ equipment and other national and transnational networks is maintained at all times.

While many people see the notion of *feature* as specific to telecommunications, I take a very broad view of features. Any part or aspect of a specification which the user perceives as having a self-contained functional

## 1.2. Features

---

role is a feature. For example, an electronic alarm clock may have features like a “snooze button” to silence the alarm for a few minutes, different beeping patterns, speaking clock, multiple alarm times, countdown timer, etc. None of these functions or features depend on each other, and they are not essential to the concept of an alarm clock.

For further illustration, some features for cars:

- in a car with electric windows, a useful feature is to have the windows close automatically when the car is locked or the alarm is activated;
- to prevent accidents the doors could automatically lock when the car travels above a certain speed<sup>1</sup>;
- the car could refuse to move unless the doors are closed and/or the seat belts are fastened (for those seats with weight on them).

From these examples it should be clear that features are units of functionality building on and enhancing the capabilities of the base system. In this general sense we can speak of *feature-oriented specification* whenever a system is specified in terms of a “base system” and features – even though this may not always be borne out by the implementation.

Thus, features are a convenient way of describing a system in terms of its functionality or capability. How this combination of functionalities is achieved, given a system and a set of additional features is a different matter. The process of adding a feature (i.e. its functionality) to a given system is what is what I call *feature integration*.

The issues that are raised by feature-oriented specification can be summarised as follows:

- how to specify a feature,
- how to add a feature to a system (feature integration),
- how to understand fully the effects of a feature on the system to which it is added (or of which it is a part), and
- how to anticipate how various features will affect each others’ function (feature interaction detection).

---

<sup>1</sup>Black taxi cabs in Britain are equipped with this “feature”.

## 1.3 Feature interaction

A feature, viewed as a change or addition to a base system, may change that system's behaviour in various ways. Both preventing existing behaviours of the base system and adding new behaviours is possible, but a feature specification will usually not describe this in a way that makes these changes easily accessible to formal analysis. Therefore, it becomes difficult to judge the effect of a feature on a system that is already extended with a feature. This difficulty is at the core of the feature interaction problem: while each feature may work fine in the absence of other features, in the presence of another feature it may not work or produce unforeseen behaviour.

A further complication is added by the fact that not all feature interactions are bad: some interactions may be considered desirable by users (or by vendors and providers), and add value to the system in question. Consider the example of an alarm clock with two alarm times and a snooze button (*cf.* previous section). If the snooze button is pressed to silence the first alarm, but the second alarm time falls within the snooze period, should the second alarm go off? Which behaviour is desirable is open to interpretation: if the second alarm does go off, this runs counter to what the user may expect of the snooze function; however if the user has set the second alarm as a “final reminder” to get up, he or she may be annoyed if that alarm does not ring at the set time. The designers of the alarm clock have to decide which of the two options they think is more sensible, or leave that choice to the user. What feature interaction analysis can do, is point out that there is a conflict between the two features.

Since features may interact both in desirable and undesirable ways, it is in general impossible to deduce whether the extended system behaves as intended, given only the specification of the base system and the features that have been added to it. In [76] Velthuisen gives a very good overview of the issues involved in the analysis of featured systems and feature interactions.

## 1.4 Reactive systems

Before we start looking at related work, it is useful to delimit the types of systems we are looking at. There are two useful distinctions for this: whether the system is centralised or distributed, and whether the system mainly performs computations or predominantly reacts to various inputs from the “environment”.

The distinction between computational and reactive systems probably needs some more explanation. I consider a software program communicating

## 1.4. Reactive systems

---

with the outside world almost exclusively through a keyboard and a screen as being rather computational, as I do every program that spends more time computing output values from input values than it does waiting for a new input, or if it outputs large amounts of data at a time. Reactive systems on the other hand, typically do not produce much output but constantly react to inputs. These reactions are usually either internal state changes and/or messages to other components, i.e. output. A typical example for a reactive system is a thermostat, whose input is the temperature at certain time intervals and whose ‘output’ is to turn the heating on or off (which may be seen as sending a message to a switch or valve).

The distinction between centralised and distributed systems is rather simpler: a system made of several or many similar components that communicate with each other to achieve their tasks is distributed, whereas a centralised system has one component that controls the rest of the system. We can understand distinction in the ‘distributed’ row as that of algorithmic parallelism versus data parallelism.

Obviously, the boundaries between these classes of systems are fuzzy, but I hope that the following table illustrates this classification with a few typical examples for each class of system.

Table 1.1: Classification of software and hardware systems

	reactive	computational
centralised	air conditioning, lift systems, conveyor belt systems	numerical applications, image processing <sup>2</sup> , type setting (e.g. <code>TEX</code> )
distributed	telephone network, electronic mail	recent code cracking efforts <sup>3</sup> , SETI@home <sup>4</sup>

As far as my treatment of features goes, I concentrate on reactive systems. Features can also be found on the computational side, but they typically have quite different characteristics. In the realm of work station (single user) software, one often finds software packages that can be extended with plug-ins (e.g. filters for image processing software). These extend the functionality

---

<sup>2</sup>excluding the graphical user interface

<sup>3</sup>e.g. <http://www.distributed.net/des/>

<sup>4</sup>SETI = Search for Extra-Terrestrial Intelligence, the SETI@home project distributes the analysis of signals from outer space to individual PCs; see <http://setiathome.ssl.berkeley.edu/>.

of the respective base system by offering new computations, e.g. graphics effects, report generation, etc.

Computational systems pose different problems as far as feature interaction analysis is concerned. Usually, unforeseen problems arise from the concatenation of computations. The aspects of timing and communication, on the other hand, tend to play a minor role. For example, plug-ins for image processing software are always invoked sequentially. Never do two ‘features’ operate on the same data simultaneously. Furthermore plug-ins tend to have a clearly defined interface to the base system, and that interface is comparatively rich, designed to allow unambiguous communication. This does not hold for typical reactive systems: communication interfaces tend to be narrow, and different parts of a reactive system may be acting on the same inputs, possibly influencing future inputs.

## 1.5 Existing approaches

The approaches to the feature interaction problem can roughly be classified by the following two criteria: “online” or “offline”, and avoidance, detection or resolution methods.

Here “offline” refers to the fact that these approaches are used before the features are deployed, often even before they are implemented (e.g. when specifying the feature). “Online” methods, on the other hand, deal with the running system.

The other criterion is whether the approach aims to prevent the occurrence of undesirable feature interactions in the first place, to detect them when they occur, or even to correct undesirable or “faulty” behaviour before it becomes a problem.

Most approaches and methods mentioned here<sup>5</sup> are domain specific in that they rely on a particular view or on the structure of the system they are designed for. Some approaches can, however, be seen as instances of more general methods, so they could serve as templates when investigating a new domain.

---

<sup>5</sup>References in the following survey are, where possible, from the proceedings of the series of International Workshops on Feature Interaction in Telecommunications and Software Systems held from 1992 to the present year [9, 19, 23, 51, 53]. (The proceedings of the first workshop [33] in the series were never published to a general audience.)

### 1.5.1 Offline detection

These methods aim to detect or anticipate feature interactions at the development stage, so that resolutions can be built in when implementing the features (and before deployment).

#### Pen and paper

Apparently, in the telecommunications industry, the prevailing approach to feature interaction detection is still “brainstorming”. Human experts look at the specifications of new features and try to think up all possible problems, and find solutions or work-arounds for them. This may be assisted by check-lists and databases of known interactions. (*Cf.* Zygan-Maus in [78, p.36f]: “In practice, teams of experts for service interaction analysis assisted by a service documentation database and possibly by a service simulation tool are a feasible solution”.) Rob van der Linden in [75] also stresses the importance of documentation and structuring the flow of information about features.

#### Static analysis

Static analysis does not investigate behaviours, executions or traces of the system but looks only at information readily (syntactically) available in the specification, such as transitions given by preconditions, actions and post-conditions, types of variables, etc. It is therefore very fast, at the expense of possibly not being very accurate. There are far too many examples to list them all in detail; I will highlight only a few.

- M. Svensson and M. Andersson [70] use static analysis to detect feature interactions in the user interface of mobile phones, i.e. ambiguous or inconsistent usage of the keypad and the display by different features;
- for the Feature Interaction Contest 2000, Mark Ryan and I used static analysis to detect feature interactions at a preliminary stage, to focus our search ([65], see section 4.5).
- part of the analysis suggested by C. Capellmann *et al.* [15] (Detection at Requirement Analysis) is definitely static analysis; they also propose model checking or testing.

#### Topological and combinatorial methods

These are in fact a subset of what I termed static analysis, but with a distinct flavour. Topological and combinatorial techniques are mainly useful in the

context of telecommunications and similar distributed systems with many similar nodes, e.g. telephone lines.

These methods look at topological aspects of feature interaction scenarios, such as how feature subscribers can be connected in telephone calls [48] or how an abstract view of a call (called a *use case*) changes through the introduction of features [4, 59]. The term *combinatorial* refers to the fact that these methods primarily aim to identify “interaction-prone call scenarios” [48] and feature combinations, thereby reducing the number of scenarios that need to be analysed in detail. Part of the work by Peng *et al.* [61] uses similar techniques.

### Logical approaches

Some of these are closely linked to combinatorial approaches in that the interaction-prone combinations are determined by certain logical conditions, e.g. the preconditions of two or more features being satisfiable simultaneously (Peng *et al.* [61]). Most logical approaches, however, use conditions of that kind as part of a test for conflicting actions or transitions and postconditions.

M. Heisel and J. Souquieres [36] use similar heuristic rules to detect potential feature interactions at the requirements stage. T. Ohta and various coauthors ([19, 47, 60, 77]) base their detection method on logical conditions for pre- and postconditions. Part of Bob Hall’s work [35, 34] uses similar rules. A. Gammelgaard and J.E. Kristensen [27] model features as constraints on a base system, and detect interactions as inconsistencies among these constraints.

Most authors use first order predicate logic, often enhanced with some notion of transition, and concentrate on a single transition step from the current to the next state. Some authors [8, 7] use temporal (predicate) logic, thereby introducing the notion of sequences of transitions. This leads directly to the next type of method used for feature interaction detection.

### Simulation and model checking

Methods using model checking and/or simulation are the most involved and most detailed ones. Here a model of the system is actually executed to detect errors or unexpected behaviour. While simulation only tries a small selection of possible behaviours, possibly chosen interactively, model checking tests all possibilities until it finds an error. Thus model checking is rather expensive in terms of computing power and memory requirements, whereas for simulation the problem lies in using the “right” sequences of actions to reveal a problem should there be one. In other words, if a simulation does

not find an erroneous behaviour, this is no guarantee that none exists. With model checking, on the other hand, if the model checker does not flag an error, we can be sure that the model is correct with respect to the properties we gave the model checker to verify.

The existence of a model is guaranteed because we explicitly give the model in a programming language. The model may be trivial, though, meaning that its transition relation is the empty or the all relation, depending on the model checking system. Avoiding trivial models can usually be achieved by performing some sanity checks with model checker. Some model checkers (e.g. FDR2) allow infinite models, in which case the verification may not terminate. In that case, a “buggy” model might only be detected when the model checker runs out of memory or the specifier runs out of patience.

Various people have used these techniques for feature interaction detection. The relevant publications have two focuses; finding properties indicative of feature interactions (or of their absence), and finding efficient ways of representing systems and their features so as to improve the performance of model checking. Both these questions lead to domain specific solutions which do not translate to other kinds of feature-oriented specification.

There is a wealth of publications in this area; I list here a selection to give an impression of the wealth of articles in this area:

- joint work with my supervisor: [65], [64], (a shorter version of this: [63]), [62]
- G. Bruns *et al.* [11], J. Kamoun and L. Logrippo [45], K. Turner: [73], [74]; B. Jonsson *et al.* [8, 44]; A.P. Felty and K.S. Namjoshi [25]; P. Gibson *et al.* [28]; R. Accorsi, L.G. Bouma *et al.* [1]; A. Khoumsi and R.J. Bevelo [49]; M. Nakamura *et al.* [58]; M. Thomas [71]; C. Capellmann *et al.* [15]
- One of the most convincing attempts to get reliable results by simulation comes from a group at Grenoble, led by F. Ouabdesslam and J.-L. Richier. In [24] they propose to direct the simulation of a featured system by probabilistic means: the specifier can define that certain choices should be made more often than others, thereby concentrating the search on interesting behaviours. This method avoids the drawbacks of pure random simulation to some degree, and does not require interactive guidance of the search. With their method, the Grenoble team won the First Feature Interaction Contest [31] in 1998.

### Artificial intelligence based methods

Finally, I mention some approaches that come from an artificial intelligence background: both D.D. Dankel II *et al.* in [22] and T. Charnois in [18] suggest the use of methods from the fields of natural language processing and knowledge representation to analyse the requirements for features, and to detect feature interactions as inconsistencies among the corresponding representations.

These approaches would be at the earliest stages of development because the requirements would be taken directly from the relevant documents rather than be formalised in some specification language first.

#### 1.5.2 Online detection

Online detection (and resolution) methods aim to avoid or detect and resolve interactions in the running system, using “feature interaction managers” or “monitors”. These typically look for anomalous behaviour (by some definition of what normal behaviour is) and then amend it, or backtrack and force the system to avoid it. One method for detecting anomalous behaviour is described in [72]: a feature monitor compares the “real” behaviour (i.e. in the presence of other features) to traces recorded with each feature in isolation. Another good example using backtracking is described in [54]: the architecture the authors use has a controlling process which “tries” each action that a feature requests for potential conflicts before committing it for execution. A further development of this work to finding resolutions to conflicts is described by S. Reiff in [66].

Some approaches based on negotiating agents have been also proposed [32], [12], [3]. These approaches aim at avoiding or resolving feature interactions. Each user is represented by a software agent which negotiates on his or her behalf to ensure that a connection between users is established that conforms to each user’s intentions. For example, if user A does not want to be disturbed, but user B wants to call A, then their agents might settle on a solution where B leaves a voice mail message. Alternatively, if B needs to speak to A personally, a ring-back could be setup, so that A calls B as soon as they are available again.

## 1.6 Verification and validation

One purpose of a specification of a system is to have a description of how an implementation of that system should behave, or what properties it should

have. Once we have an implementation, we want to check that it does indeed conform to the specification.

There are various languages for specifying systems, each with its strong points. Many specification languages are specialised to certain applications, e.g. hardware and chip design, communication protocols, etc.

Many specification languages have associated tools, such as simulators or model checkers. This section will give a brief overview of some specification languages and associated tools.

### 1.6.1 Specification languages

Specifications can take several forms. One can describe a system in terms of its (intended) properties, or in terms of its composition and structure. Furthermore, a specification may be declarative, i.e. stating what the system does, or functional, i.e. saying how it does it. Most specification formalisms allow the user to express these different aspects by offering modularisation and composition of (sub-)specifications. Especially for functional aspects, however, it is helpful to have a “property language” to express *requirements* which every implementation of the functional description should satisfy. This is where, apart from first order logic, very often temporal logics are used.

### 1.6.2 Temporal logics

Temporal logics come in many different flavours. I will concentrate only on the most wide-spread variants. All of the following logics are commonly interpreted over Kripke structures [30, 69], however the algorithms used for model checking differ.

- CTL (Computation Tree Logic)
- CTL\* (extended CTL)
- LTL (Linear Time Temporal Logic)
- modal  $\mu$ -calculus

#### CTL\*

CTL\* (full Computation Tree Logic) can express all temporal properties that rely only on *states*.

Given a set  $P$  of propositional atoms, CTL\* formulas are given by the following syntax:

$$\begin{aligned}\varphi &::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{A}\psi \mid \mathbf{E}\psi \mid \\ \psi &::= \top \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \mathbf{X}\psi \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \psi_1 \mathbf{U}\psi_2\end{aligned}$$

where  $\varphi$  describes state formulas,  $\psi$  path formulae, and  $p \in P$ . The other boolean operators ( $\vee, \rightarrow, \leftrightarrow, \perp$ ) are defined in terms of  $\wedge, \neg$  and  $\top$  in the usual way. In state formulas,  $\mathbf{A}$  means ‘along all paths beginning in the current state’ (*inevitably*), and  $\mathbf{E}$  means ‘along at least one path from the current state’ (*possibly*). For path formulas  $\mathbf{X}$ ,  $\mathbf{F}$ ,  $\mathbf{G}$ , or  $\mathbf{U}$  mean ‘neXt state’, ‘some Future state’, ‘all future states (Globally)’ (along the path under consideration), and ‘Until’ respectively. Notice that  $\mathbf{U}$  is binary.

## CTL

As the name Computation Tree Logic suggests, CTL allows us to express properties of branching time. CTL is the branching time fragment of CTL\*.

Given a set  $P$  of propositional atoms, CTL formulas are given by the following syntax:

$$\begin{aligned}\varphi &::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \\ &\quad \mathbf{A}\mathbf{X}\varphi \mid \mathbf{E}\mathbf{X}\varphi \mid \mathbf{A}\mathbf{G}\varphi \mid \mathbf{E}\mathbf{G}\varphi \mid \mathbf{A}\mathbf{F}\varphi \mid \mathbf{E}\mathbf{F}\varphi \mid \mathbf{A}[\varphi_1 \mathbf{U}\varphi_2] \mid \mathbf{E}[\varphi_1 \mathbf{U}\varphi_2].\end{aligned}$$

where  $p \in P$ . The other boolean operators ( $\vee, \rightarrow, \leftrightarrow, \perp$ ) are defined in terms of  $\wedge, \neg$  and  $\top$  in the usual way.

Notice that CTL temporal operators come in pairs so that we only get state formulas. The first of the pair is one of  $\mathbf{A}$  and  $\mathbf{E}$ . The pair of operators in  $\mathbf{E}[\varphi_1 \mathbf{U}\varphi_2]$ , for example, is  $\mathbf{EU}$ . Further details of CTL are widely available in the papers by E. Clarke and others [21, 55], and also in the introductory text [41].

Two useful derived connectives are  $\mathbf{AW}$  and  $\mathbf{EW}$ , which use the ‘weak until’ connective  $\mathbf{W}$ , which is similar to  $\mathbf{U}$ , but  $\varphi_1 \mathbf{W}\varphi_2$  does not require that  $\varphi_2$  eventually becomes true if  $\varphi_1$  is indefinitely true. One defines  $\mathbf{A}[\varphi_1 \mathbf{W}\varphi_2]$  as  $\neg \mathbf{E}[\neg\varphi_2 \mathbf{U}\neg(\varphi_1 \vee \varphi_2)]$ , and  $\mathbf{E}[\varphi_1 \mathbf{W}\varphi_2]$  as  $\mathbf{E}[\varphi_1 \mathbf{U}\varphi_2] \vee \mathbf{EG}\varphi_1$ .

## LTL

Linear Time Temporal Logic expresses properties about sequences of states, rather than about trees. LTL is the linear time fragment of CTL\*.

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \square\varphi \mid \diamond\varphi \mid \varphi_1 \mathbf{U}\varphi_2 \mid \varphi_1 \vee \varphi_2$$

$\bigcirc\varphi$  holds iff  $\varphi$  holds in the *next* state along the path,  $\square\varphi$  iff it always holds and  $\diamond\varphi$  if it holds eventually, i.e. at some future state on the path. The until operator  $U$  is the LTL version of CTL\*  $U$ . Some authors also use the CTL\* notation for LTL and write  $\mathbf{X}$ ,  $\mathbf{G}$  and  $\mathbf{F}$  instead of  $\bigcirc$ ,  $\square$  and  $\diamond$ , respectively.

LTL formulas describe sets of infinite strings over the set of atomic propositions  $P$ , which are in fact  $\omega$ -regular. Since  $\omega$ -regular languages are recognised by Büchi automata, these are used for LTL model checking.

### 1.6.3 Model checking

Model checking is a method of automatic verification by which a specification is shown to have certain generic or specific properties. While there are some approaches to deal with certain classes of infinite state systems<sup>6</sup>, I will only describe the finite state case.

For any model checker, the model to be checked is a finite state machine, usually given in some programming language. The properties can be generic, e.g. deadlock-freedom, or given in a logic, e.g. invariants and liveness properties, or they can be relations between two such models, e.g. refinement or bisimulation. Depending on whether relations between models or properties of one model are to be verified, there are two ways of phrasing the verification problem.

For the property-oriented approach the model checker does the following: given a model  $S$  and a property  $\varphi$ , the model checker explores the state space of the model  $S$  to find a violation of  $\varphi$ . Note that  $\varphi$  will, in general, not be a predicate on states (a state invariant), but e.g. a temporal logic formula or some other specification of behaviour over time. A typical example of a property checker is SMV, which is described in the next chapter. In Chapter 3, I will introduce SPIN which falls in this category, too.

For the relation-oriented approach, the model checker is given two models (often, one is taken to represent the intended behaviour) and the relation to be verified. The model checker then executes the models in parallel, i.e. it explores the product of their state spaces, looking for pairs of states or transitions in the execution that violate the relation. The CSP model checker FDR2, which we will see in Chapter 4, is a typical example of a relation-based model checker.

SMV<sup>7</sup> was originally designed for describing hardware, especially clocked circuits, and it uses a declarative language for describing the system, and

---

<sup>6</sup>Such methods usually rely on ways to factorise the state space into finitely many classes, which can then be treated with finite state algorithms, e.g. Henzinger in [37].

<sup>7</sup>Symbolic Model Verifier

CTL to express desired properties of the specified system. The SMV model checker verifies whether the properties hold for the system specified.

PROMELA<sup>8</sup> is an imperative language, designed to describe communication protocols, and properties to be verified can be stated in a LTL. The model checker SPIN verifies that a PROMELA program satisfies a given LTL formula; it can also check for deadlocks and certain livelocks.

CSP<sup>9</sup> is a declarative language for describing processes and their communication. FDR2<sup>10</sup> is the model checker for CSP; it can check certain generic properties of a system (deadlock, livelock, nondeterminism) and check the refinement relation between two systems.

#### 1.6.4 Simulation and testing

The terms simulation and testing are often used almost synonymously. First and foremost, both methods do not explore all possible executions or traces of any given system. So they can in general only give some confidence in the correctness of a system under investigation, never certainty. As mentioned before, this is the tradeoff for being able to analyse far more complex systems than would be possible by model checking. Testing is usually concerned with complete systems, but it can of course be applied to an executable model of a system, too.

The distinction between testing and simulation is that simulation can be seen as one method used in testing. However, testing also comprises a whole theory of the level of confidence to which a given set of tests ascertains that the tested system is indeed correct with respect to the specification. An important concept in testing is *coverage*, that is how much of the space of possible behaviours is explored. The different ways of measuring *coverage*, and how this relates to confidence, is one of the subjects of testing theory.

For the purposes of feature interaction detection, we can simply take testing as a given set of methods, and use it when, e.g. model checking fails due to the complexity of the task. It is worth noting, though, that the application to feature interaction detection changes the parameters of how to design and assess the quality of a test suite. After all, we assume that the base system is correct, and a featured system preserves this as long as no features are activated. Thus, one wants to concentrate on test scenarios that exercise the features.

---

<sup>8</sup>PROMELA stands for Protocol Modelling Language; SPIN is not an acronym.

<sup>9</sup>Communicating Sequential Processes

<sup>10</sup>FDR stands for Failures-Divergences Refinement, a semantics for CSP.

## 1.7 Specifying features

In this section I will give an outline of how features could be specified using special constructs added to a specification language. This and the next section describe the programme of my research in general terms.

### 1.7.1 Feature constructs

Since we understand complex systems by a kind of functional decomposition, we would like to reflect that in the way that we specify features, i.e. functional components. Usually we can identify the basic or core functionality of a system and separate out the nonessential functionality. In other cases we are given the basic system and want to add new, nonessential functionality. Either way, despite being perceived as functional units, features are often *not* encapsulated in modules, functions or procedures, the traditional structures for functional decomposition. Instead, features tend to be pervasive, affecting or at least drawing on many parts of the base system.

Therefore we need a new way of structuring the system, one that corresponds to our conceptualisation of a base system plus a set of features. This leads to the idea of a *feature construct*: a language construct that is like a module in a traditional language in that it encapsulates certain functionality, but whose semantics make it affect all relevant parts of the system to which it is added. The design principles for feature constructs are simple: features should be

- easy to specify,
- easy to understand,
- easy to add or remove,
- easy to revise.

Specifying features as textual units goes some way to meet these goals: all the necessary information is concentrated in one place, except for the parts of the base system that are used.

In addition to this, a feature construct should fit in with the syntax of the specification language in question. This will make it easier to learn. Moreover, the entities of the base system that a feature will inevitably refer to are defined in that language, thus using a similar syntax for defining features will look more natural.

## 1.7.2 Feature integration

Since we are often faced with the problem of defining and adding new features to existing system specifications<sup>11</sup>, it makes most sense to define the semantics of features by syntactic manipulation of the base specification.<sup>12</sup> If one were to make a change in the semantics in the underlying specification language, it would be difficult to establish the relationship with the original semantics. In case of an executable language, like the three languages we will see in the remainder of this thesis, one would also need to rewrite the compiler or model checker to implement the new semantics. By adding features through purely syntactic means, we can avoid these problems.

The preprocessor or front-end that performs these syntactic changes on the base program according to the feature definition is called *feature integrator*. A feature integrator generally takes as input a base system in the given language and one or more features defined in the extended syntax of the feature construct and outputs a new specification of the featured system which uses only the specification language.

The resulting specification can then be analysed using the same off-the-shelf methods and tools as the base system.

## 1.7.3 Model checking approach to feature interactions

The use of model checking for feature interaction detection is in being able to add a set of features to a given base system, and then automatically check the resulting system for abnormal behaviour. The properties to be checked may be either generic, such as deadlock freedom, or specific to the base system and/or the features, to test whether the featured system operates as expected. We shall generally take the view that a feature comes with a set of properties that a system with the feature is expected to comply with.

## 1.7.4 Related work

The following ideas are in some ways similar to our notion of a feature construct, although the first two come from rather different areas and are aimed at somewhat different problems.

- S. Katz's concept of *superimpositions* [46] is quite similar to the idea of features, and the way Katz proposes to specify superimpositions

---

<sup>11</sup>As an extreme case, the source code of a program may be seen as a specification of the program.

<sup>12</sup>Both superimpositions and aspect-oriented programming (see section 1.7.4 on related work below) are similar to the feature construct approach in this respect.

corresponds very closely to the use of a feature construct. Both approaches allow one to add functionality to an existing system, and in both one specifies the added functionality in a separate textual unit like a module. Finally the resulting system description is generated by integrating the system description with the specification of the feature or superimposition. The similarity is especially strong for the feature construct for PROMELA (see chapter 3). PROMELA is an imperative language, offering concurrency through processes executing (virtually) in parallel. This is similar to Katz’s setting.

Examples of uses for superimpositions are monitoring the base system, and ensuring termination of a collection of parallel processes when some suitable end state has been reached.

- aspect-oriented programming (AOP) [50]

The authors define an aspect as “a property that must be implemented [...] if it cannot be cleanly encapsulated in a generalised procedure [i.e. function, object, etc.]. Aspects tend not to be units of the system’s functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronisation of concurrent objects.”

The crucial property of aspects is that they “cut across” the functional decomposition of a system, and because of that they affect many functions, modules or objects<sup>13</sup>. This is also often true of features. The view that AOP takes, however, sees the component language and the aspect language as more or less orthogonal to each other, one dealing with the “what” the other with the “how” of a solution to a problem. Feature-oriented specification, on the other hand, sees the original system as a basis to build on, as defining some domain and a set of actions one might like to perform on or within that domain. The features refine or redefine parts of that system, both in the “what” and the “how” sense.

However, features usually implement additional functionality that the base system did not provide. It is the addition of such “functions” to an existing system that often requires changes to many parts of the system — and in this sense features cut across the original structure.

What Kiczales and coauthors write in [50] about implementing systems in AOP echoes arguments for feature-oriented specification: “While

---

<sup>13</sup>Whatever the structures used for the functional decomposition are.

asking the programmer to explicitly address implementation aspects sounds like it might be a step backwards, our experience with work on open implementation suggests that in fact it isn't. While the programmer is addressing implementation in the memory aspect, proper use of AOP [Aspect-Oriented Programming] means that they are expressing implementation strategy at an appropriately abstract level, through an appropriate aspect language, with appropriate locality. They are not addressing implementation details, and they are not working directly with the tangled<sup>14</sup> implementation.”

- Brederke in [10] suggests structuring specifications or requirements documents in families, possibly with a hierarchical structure. Some aspects of [10] come close to the idea of a feature construct, in that he extends a specification language (CSP-OZ) to allow non-monotonic composition of specifications. So far, tool support for Brederke's method is only sketched, but it seems very similar to our approach, using a feature integration tool and then generic validation tools such as a type checker and a model checker on the featured system.

## 1.8 The rest of this thesis

In this thesis, I will explore

- the use of feature constructs for defining features,
- the potential of feature constructs for detecting and analysing feature interactions,
- the usefulness of model checking for feature interaction detection,
- and – mainly in chapter 4 – the use of static analysis for the detection of feature interactions.

All this will be in the context of offline analysis, aiming at the specification stage in the software life-cycle. My general framework always is an existing specification language for which there is a model checker. I extend these model checking languages with suitable feature constructs and explore their expressiveness as well as the capabilities of the model checker with respect to the objective of validating featured systems.

The aim of this exercise is to demonstrate that a feature construct (as an extension of an existing language) is a useful tool for defining and analysing

---

<sup>14</sup>“Tangling” is the AOP equivalent of feature integration.

features and their interactions. At the beginning of my studies I was expecting model checking to be a good method to detect feature interactions. However, we will see that model checking fails to meet these expectations in general. There are some avenues of research – which I did not have the time to investigate – that may temper the negative verdict on model checking; I mention these in the conclusions to the following chapters.

I hope to convince the reader, however, that feature constructs have a lot to offer for the specification and analysis of features, and that they may be of help in feature interaction detection.

Feature constructs offer a good way of structuring complex systems, thus helping to understand them and focusing on the relevant parts when looking for feature interactions. For tool-based methods, such constructs can offer a good formal framework.

### 1.8.1 Outline

The structure of this thesis is as follows: The next three chapters deal with three different instantiations of the feature construct idea. In each chapter I choose a different model checking language, define a suitable feature construct for it and demonstrate its use in some examples.

The three feature constructs may look quite different, but they are driven by the same ideas: a feature *requires* certain things of the base system, it *introduces* new objects to the system, and – and this is the essence of a feature – it *changes* things in the system. How these actions are specified, however, depends very much on the language we extend. SMV has a declarative flavour, PROMELA is an imperative language, while CSP has a functional character. This calls for different realisations of the feature construct idea in each case. Especially the *change* aspect of features needs to be adapted to the key elements of each language: for SMV, these are assignments to variables; in PROMELA, it is also flow control and communication; while in CSP, there are no assignments, except as a result of communication, so communication is one central component, the other one being the composition of processes.

The chapters on SMV (Chapter 2) and the one on CSP (Chapter 4), define formal semantics for the respective feature constructs, above and beyond the indirect semantics given by the feature integration process. These semantics give us a way to reason formally about features in addition to mere experimentation. In the final chapter I sum up the lessons learnt and draw general conclusions about the use of feature constructs and model checking, indicating worthwhile areas for further research. It will turn out that static analysis may be a more effective means for the detection of feature interactions than model checking.

**Statement of originality**

The work and results presented in this thesis are mine unless stated otherwise. I have taken all due care to indicate other people's contributions where applicable.

# Chapter 2

## Features for SMV

### 2.1 Introduction

This chapter revolves around the SMV<sup>1</sup> model checker.<sup>2</sup> I give a brief introduction to the input language, also called SMV, before I describe a feature construct for this language. I then demonstrate the use of this feature construct in two case studies, a lift system and a network of telephones. This is followed by a theoretical treatment of the feature construct: I develop the denotational semantics of features and prove some results about non-interaction of features.

The work described in this chapter is joint work with my supervisor, Mark Ryan, who developed the notion of a feature construct for SMV as well as the syntax. When I started, Mark Ryan had developed the basic aspects of the feature construct. Jointly we defined the exact meaning of the `TREAT` and `IMPOSE` clause, which I implemented in the SMV feature integrator.

We worked closely together both on the case studies and the semantics. The base system for the lift was taken from Mark Berry's MSc thesis [6], while the phone system resulted from merging a synchronous model designed by Mark Ryan with an asynchronous one which I developed. (The asynchronous character of the latter proved problematic in verification with SMV.) We chose the features to model from the relevant literature. All features were coded by me.

---

<sup>1</sup>Symbolic Model Verifier

<sup>2</sup>Until 1998 there was just one SMV, but now there are three. CMU SMV [55] is the original one, developed by Ken McMillan, and is the one I use in this thesis. NuSMV is a re-implementation being developed in Trento [20], and is aimed at being customisable and extensible. Cadence SMV is an entirely new model checker focused on compositional systems. It is also developed by Ken McMillan, and its description language resembles but much extends the original SMV [56].

Mark Ryan developed the semantics for SMV and the feature construct which I present in section 2.5, and proved the theorems presented. However, I generalised the theorems somewhat, removing some of the constraints, especially where the use of the next operator is concerned.

## 2.2 The SMV model checker

This exposition of SMV is kept very brief and I refer the interested reader to [21, 55] for a more detailed account.

SMV is a verification tool which takes as input

- a system description in the SMV language, and
- some formulas in the temporal logic CTL.

SMV produces as output the statement ‘true’ or ‘false’ for each of the formulas, according to whether the system description satisfies the formula or not. In symbols we write  $S \models \varphi$  if the system  $S$  satisfies the formula  $\varphi$ , and  $S \not\models \varphi$  if  $\varphi$  does not hold for  $S$ . Additionally, SMV produces a counterexample for each universal formula (i.e. one starting with **AG** or **AF**) that is not satisfied, or a witness for each existential formula that is satisfied.<sup>3</sup>

### 2.2.1 The SMV language

The SMV description language is essentially a high-level syntax for describing finite state automata. It provides modularisation, and synchronous and asynchronous composition. The behaviour of the environment is modelled by non-determinism. An SMV system description declares the state variables, their initial values and the next values in terms of the current and next values of the state variables – as long as this does not lead to circular dependencies.

SMV works with unlabelled automata and has no message passing. Hence all synchronisation has to be by explicit references to current and next values. While this keeps the syntax simple, it does sometimes make writing the description slightly cumbersome.

Figure 2.1 shows one of the examples distributed with the SMV system. (The line numbers are not part of the code.) This piece of code defines an automaton with four states ( $\{0, 1\} \times \{ready, busy\}$ ). There are transitions from every state to every state, except for the state  $(1, ready)$  from which only transitions to  $(1, busy)$  and  $(0, busy)$  are allowed. The initial states are  $(1, ready)$  and  $(0, ready)$ . In Figure 2.2 you can see the diagram of the au-

---

<sup>3</sup>For formulas with both existential and universal quantifiers, SMV’s output is usually not very helpful.

```

1: MODULE main
2: VAR
3:   request : boolean;
4:   state : {ready,busy};
5: ASSIGN
6:   init(state) := ready;
7:   next(state) := case
8:     state = ready & request : busy;
9:     1 : {ready,busy};
10:   esac;
11: SPEC AG(request -> AF state = busy)

```

Figure 2.1: A system description for SMV

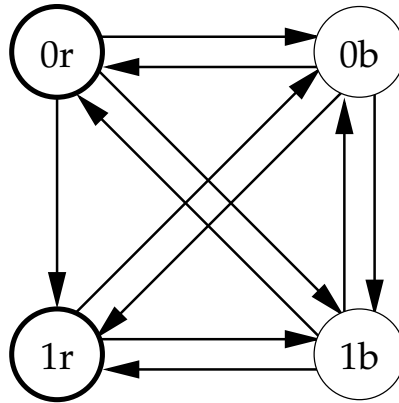


Figure 2.2: Transition diagram

tomaton defined by the SMV code in Figure 2.1. Note that SMV interprets assignments stated in the program as *constraints* on the automaton with all transitions. Any variable whose value is not determined by an assignment will assume values non-deterministically. Hence in the example, the transition from  $(1, \text{ready})$  to  $(0, \text{ready})$  is not present: the first case of the assignment to state rules out this transition. All other transitions remain allowed.

Generally, a model description for SMV consists of a list of modules with parameters. Each module may contain variable declarations (**VAR**), macro definitions (**DEFINE**), assignments (**ASSIGN**), and properties (**SPEC**) to be checked of the module. Every SMV program must at least define the module **main**.

Possible types for variables are boolean ( $\{0, 1\}$ ), enumeration types (e.g.

`state`), finite ranges of integers, or arrays of these types. For declared variables (as opposed to `DEFINED` ones, which are merely macros) we may assign the initial value (e.g. line 6) and the next value (e.g. lines 7–10), or alternatively, the current value. The expressions that are assigned to variables may be non-deterministic as in line 9: if `state` is not *ready* or `request` is 0, the next value of `state` can be either *ready* or *busy*. (Since `request` is not determined at all by the description, it too will assume values non-deterministically.) Note that `case` statements are evaluated top to bottom, so the result is the expression from the first branch whose condition evaluates to true. It is important to bear in mind that all assignments are evaluated in parallel. (There is also a mechanism for asynchronous (interleaving) composition of modules but I do not use it, not least because its implementation in CMU SMV showed displayed strange behaviour.)

A special kind of variable declaration is the instantiation of a module, as in “`b1 : button(floor=1);`”. This is interpreted as a declaration of all local variables (including `DEFINED` identifiers) of that module, prefixed with the name of the newly declared variable, together with the assignments and macro definitions within that module. If the module `button` has a local variable `pressed`, the declaration above implicitly declares `b1.pressed`. The formal parameters are replaced by the actual parameters as in a call-by-name language.

It is possible to assert fairness constraints on the model. These are given by propositional formulas preceded by the keyword `FAIRNESS`. In the presence of such fairness constraints, only executions are considered along which these constraints are true infinitely often.

After defining a system in the SMV language, we formulate the properties to be verified in the temporal logic CTL (marked by the keyword `SPEC`, e.g. line 11 in Figure 2.1 asking SMV to verify that in any state (“`AG`”) it holds that whenever `request` is true then, eventually, (“`AF`”) `state` will have the value `busy`). The propositional atoms for these formulas are the boolean variables and the equations over the variables and constants of the system.

## 2.3 A feature construct for SMV

In this section I describe the SMV feature construct and show how model descriptions written using the feature construct can be integrated into an SMV program, thus giving semantics to the feature construct. Following from that I will illustrate its use in two case studies.

```
FEATURE feature-name
[ REQUIRE
  { MODULE module-name [ (parameter-list) ]
    VAR variable-declarations }*
]
[ INTRODUCE
  { MODULE module-name
    [ VAR variable-declarations ]
    [ ASSIGN assignments ]
    [ DEFINE definitions ]
    [ { SPEC formula }* ] }*
]
[ CHANGE
  { MODULE module-name
    [ IF condition THEN ]
      [ IMPOSE assignments
        | TREAT var1 = expr1 [ , ... varn = exprn ]
      ] }*
]
END
```

where: [ ] stands for ‘optional’  
[ | | ] stands for ‘one of’  
{ }\* stands for ‘several’

Figure 2.3: The syntax of the feature construct

### 2.3.1 Syntax of the feature construct

A formal specification of the syntax of the feature construct is given in Figure 2.3. There are three main sections of the feature construct, introduced by the keywords `REQUIRE`, `INTRODUCE` and `CHANGE`.

The `REQUIRE` section stipulates what entities are required to be present in the base program in order for the feature to be applicable. A collection of modules and variables in modules may be specified there. All modules and variables of the base system that are used in the `INTRODUCE` and `CHANGE` sections should be `REQUIRED`, and their absence will lead to an error.

The `INTRODUCE` section states what new modules or new variables within old modules are introduced by the integration of the feature into a program. `DEFINE` and `ASSIGN` clauses may also be given, as may CTL formulas in `SPEC`. These are textually added to the SMV text during feature integration.

The **CHANGE** section specifies how the feature changes the behaviour of the system with respect to the original state variables. It gives a number of **TREAT** or **IMPOSE** clauses, which may be guarded by a condition. This is where the behaviour of the original system is altered.

The idea of **TREAT** and **IMPOSE** is to alter how a transition from one state to another is realised in the model: by reading the values of (current) state variables and assigning new values (computed from the current values) to state variables for the next state. **TREAT** changes the value read from a variable, **IMPOSE** changes the value assigned to a variable. This is appropriate for SMV, since state variables and assignments are the only means of defining transitions<sup>4</sup>

### 2.3.2 Integrating a feature

Given an SMV text representing the base system, and a feature description, our integration tool SFI does the following:

- It checks that the **REQUIRED** entities are present in the base system, and reports an error if they are not.
- It inserts the SMV code for the new modules and variables declared in the **INTRODUCE** section.
- For **CHANGES** of the form  
`IF cond THEN TREAT x = expr`  
it replaces all right-hand-side occurrences of *x* by

```
case
  cond : expr;
  1     : x;
esac
```

This means that whenever *x* is read, the value returned is not *x*'s value, but the value of this expression. Thus, when *cond* is true, the value returned is *expr*. In short, when *cond* is true, we treat *x* as if it had the value given by *expr*. Note that we require *expr* to be deterministic because *x* may occur in conditions in other case statements in the base system, and SMV requires such conditions to be deterministic.

---

<sup>4</sup>SMV offers the possibility to define a transition relation explicitly; this is the equivalent to machine language programming. However, the idea of a feature construct is aimed at (relatively) high-level languages.

### 2.3. A feature construct for SMV

---

- For **CHANGE**s of the form

**IF** *cond* **THEN** **IMPOSE**  $x := expr$ ;

In assignments  $x := oldexpr$  or  $next(x) := oldexpr$ , the integrator replaces *oldexpr* by

```
case
  cond : expr;
  1    : oldexpr;
esac
```

Whereas **TREAT** just deals with expressions reading the value of  $x$ , i.e. occurrences of  $x$  on the right-hand-side of an assignment to another variable, **IMPOSE** deals with assignments to the variable  $x$ . It has the effect that, when *cond* is true,  $x$  is assigned the value of *expr*; but when *cond* is false,  $x$  is assigned the value that it would have been assigned in the original program. In an **IMPOSE** statement, *expr* may be non-deterministic.

**Remark:** For **CHANGE**s that are not guarded by **IF** *cond* **THEN**, the **case** statements can be omitted, and the variable ( $x$ ), or respectively, the expression (*oldexpr*), is replaced directly by the new expression (*expr*). Moreover, any **TREAT** feature can be rewritten such that the conditional is part of the expression *expr*, since right-hand-side occurrences of  $x$  are replaced by a conditional (**case**) expression guarded by *cond* anyway.

That is, **IF** *cond* **THEN** **TREAT**  $x = expr$  is equivalent to:

```
TREAT  $x =$  case
  cond : expr;
  1    :  $x$ ;
esac
```

The integration of a feature is deemed successful if the following are true:

- The modules and variables stipulated in the **REQUIRE** section were present in the base program; and
- After the textual substitutions have been performed, the resulting program satisfies the CTL formulas in the **INTRODUCE** section of the feature.

Notice that in general one cannot expect the CTL formulas of the base system to hold, since the feature was introduced to alter the behaviour of the system.

### 2.3.3 Integration of multiple features

When several features are integrated in succession, the question arises if and how the order of integration matters. From the explanations above, it is clear that the order of integration does matter in general. The details of how the features affect each other are quite complicated however. As a rule of thumb, one can assume that features which are integrated *later* take precedence over features integrated previously.

In section 2.4 we explore feature integration in the context of our case studies. This will illustrate the effect of integrating features in different orders.

### 2.3.4 Detecting feature interaction

One can view a feature as comprising two components: the feature implementation and the feature properties. In the following I write  $(F, \varphi)$  for a feature when I want to stress this distinction. In practice it is usually more useful to state the requirements as several formulas. The formula  $\varphi$  then stands for a specific property one would like to verify of the system. When we integrate a feature  $(F, \varphi)$  into a base system  $S$ , to yield a new system  $S + F$ , we want to test the following:

- $S + F \models \varphi$ : Feature  $F$  has been successfully integrated.
- $(S + F_1) + F_2 \models \varphi_2$ : Feature  $F_2$  can be integrated into the extended system  $S + F_1$ .
- $(S + F_1) + F_2 \models \varphi_1$ : Feature  $F_2$  does not violate the requirements of  $F_1$ .

Of course these tests will not necessarily succeed. For the remainder of this section, we shall however assume that all features are correct with respect to the base system, i.e.  $S + F \models \varphi$  for any feature  $(F, \varphi)$ . Then we can observe feature interaction in the following forms:

- Type I:  $(S + F_1) + F_2 \not\models \varphi_2$ :  
Earlier feature breaks later one.
- Type II:  $(S + F_1) + F_2 \not\models \varphi_1$ :  
Later feature breaks earlier one.
- Type III:  $S, S + F_1, S + F_2 \models \psi$  but  $(S + F_1) + F_2 \not\models \psi$ :  
(where  $\psi$  is a property of the base system.)  
Features combine to break system.

- Type IV:  $\exists \varphi. (S + F_1) + F_2 \models \varphi$  but  $(S + F_2) + F_1 \not\models \varphi$ :  
(where  $\varphi$  is a property of  $S$ ,  $F_1$  or  $F_2$ )  
Features do not commute.

Note that these types of interactions do not represent a disjoint classification; two features may exhibit several types of interaction. Obviously, for commuting features, a Type I interaction for integration of  $F_1$  and then  $F_2$  corresponds to a Type II interaction for the reverse order of integration, and vice versa. I will come back to this classification in the analysis of our case studies.

Other authors, e.g. Hall [35] and Cameron, Griffeth *et al.* [14], also categorize feature interactions. However most of these classifications are dependent on a specific method or view of the system; my classification is based purely on properties.

## 2.4 Case studies

To test the usefulness and expressiveness of the SMV feature construct, Mark Ryan and I have conducted two case studies. The lift system that I describe in the following section is a typical example of a centralised reactive system; the telephone system in section 2.4.2 represents a distributed reactive system.

### 2.4.1 A lift system with five features

#### The basic lift system

As a first case study, we analysed a *lift system* and its features. For the base system we adapted the lift system description written by Mark Berry [6]. The SMV code for a single lift travelling between 5 floors is given in Figures 2.4 to 2.5. It consists of about 120 lines of SMV code.

The module `main` (Figure 2.4) declares five instances (one for each landing) of the module `button` (passing to each one as argument the conditions under which that button should reset itself). It also declares one instance of `lift`, to which it passes two parameters: (Recall that the parameters are treated in a call-by-name fashion.)

- `landing_call` holds the number of the next landing in the current direction of travel at which there was a request for the lift,
- and `no_call` is true when there is no landing request (regardless of current direction). Note that `no_call` is *not* equivalent to `landing_call` being 0.

```
MODULE main
VAR
  landingBut1 : button ((lift.floor=1) & (lift.door=open));
  landingBut2 : button ((lift.floor=2) & (lift.door=open));
  landingBut3 : button ((lift.floor=3) & (lift.door=open));
  landingBut4 : button ((lift.floor=4) & (lift.door=open));
  landingBut5 : button ((lift.floor=5) & (lift.door=open));

  lift      : lift (landing_call, no_call);

DEFINE
landing_call:=
  case
    lift.direction = down :
      case
        landingBut5.pressed & lift.floor>4 : 5;
        landingBut4.pressed & lift.floor>3 : 4;
        landingBut3.pressed & lift.floor>2 : 3;
        landingBut2.pressed & lift.floor>1 : 2;
        landingBut1.pressed                : 1;
        1                                   : 0;
      esac;
    lift.direction = up   :
      case
        landingBut1.pressed & lift.floor<2 : 1;
        landingBut2.pressed & lift.floor<3 : 2;
        landingBut3.pressed & lift.floor<4 : 3;
        landingBut4.pressed & lift.floor<5 : 4;
        landingBut5.pressed                : 5;
        1                                   : 0;
      esac;
  esac;

no_call := (!landingBut1.pressed &
           !landingBut2.pressed &
           !landingBut3.pressed &
           !landingBut4.pressed &
           !landingBut5.pressed);
```

Figure 2.4: The SMV code for the module `main` in the lift system.

## 2.4. Case studies

---

```
MODULE button (reset)
VAR
  pressed : boolean;
ASSIGN
  init (pressed) := 0;
  next (pressed) := case
    reset      : 0;
    pressed    : 1;
    1          : {0,1};
  esac;
```

Figure 2.5: The SMV code for the module `button` in the lift system.

The module `main` only determines the values of these two variables, everything else is delegated to the `lift` module (Figure 2.6) and the five instances of the `button` module (Figure 2.5).

The `lift` module (Figure 2.6) declares the variables `floor`, `door` and `direction` as well as a further 5 buttons, this time those inside the lift. The algorithm it uses to decide which floor to visit next is the one called “Single Button Collective Control” (SBCC) from [5]: the lift travels in its current direction answering all lift and landing calls until no more exist in the current direction; then it reverses direction, and repeats. Actually the conditions under which it reverses direction are slightly more complicated, as can be seen by inspecting the code for `next(direction)` in Figure 2.6: if the lift is idle, it maintains the same direction as it had before, but if it is at the top or bottom of its shaft it changes direction to down and up respectively; otherwise, as stated, it reverses direction if there are no calls remaining to be served in the current direction. The final ‘`1:direction`’ means that if none of the preceding conditions are true, then the value returned by the case statement is simply the old value of `direction`. Notice that the SBCC algorithm stipulates only one button on each landing, rather than the conventional two. Passengers press the button, but they are not guaranteed that the lift will be willing to go in the direction they wish to travel.

By inspecting the `button` module (Figure 2.5), one finds that its variable `pressed` is set to false if the `reset` parameter is true; otherwise, if it was pressed before, it persists in that state; otherwise, it non-deterministically becomes true or false. This non-determinism is to model the fact that a user may come along and press the button at any time. In common with most actual lift systems, the user may not un-press the button; once pressed, it remains activated until the conditions to reset it arise inside the lift system.

```

MODULE lift (landing_call, no_call)
VAR
  floor      : {1,2,3,4,5};
  door       : {open,closed};
  direction  : {up,down};
  liftBut5   : button (floor=5 & door=open);
  liftBut4   : button (floor=4 & door=open);
  liftBut3   : button (floor=3 & door=open);
  liftBut2   : button (floor=2 & door=open);
  liftBut1   : button (floor=1 & door=open);

DEFINE
  idle      := (no_call & !liftBut1.pressed & !liftBut2.pressed &
               !liftBut3.pressed & !liftBut4.pressed & !liftBut5.pressed);
  lift_call :=
    case
      direction = down :
        case
          liftBut5.pressed & floor>4 : 5;
          liftBut4.pressed & floor>3 : 4;
          liftBut3.pressed & floor>2 : 3;
          liftBut2.pressed & floor>1 : 2;
          liftBut1.pressed           : 1;
          1                           : 0;
        esac;
      direction = up   :
        case
          liftBut1.pressed & floor<2 : 1;
          liftBut2.pressed & floor<3 : 2;
          liftBut3.pressed & floor<4 : 3;
          liftBut4.pressed & floor<5 : 4;
          liftBut5.pressed           : 5;
          1                           : 0;
        esac;
    esac;

ASSIGN
  door := case
    floor=lift_call      : open;
    floor=landing_call   : open;
    1                    : closed;
  esac;

  init (floor) := 1;
  next (floor) := case
    door=open           : floor;
    lift_call=0 & landing_call=0 : floor;
    direction=up & floor<5      : floor +1;
    direction=down & floor>1     : floor -1;
    1                    : floor;
  esac;

  init (direction) := down;
  next (direction) := case
    idle                : direction;
    floor = 5           : down;
    floor = 1           : up;
    lift_call=0 & landing_call=0 & direction=down : up;
    lift_call=0 & landing_call=0 & direction=up   : down;
    1                  : direction;
  esac;

```

Figure 2.6: The SMV code for the module `lift` in the lift system.

### Properties for the basic lift system.

Before any features are added, we may use SMV to check basic properties of the lift system. For example, the following CTL specification in the module `main` is satisfied: *pressing the button on landing  $i$  guarantees that the lift will arrive at that landing and open its doors.* In CTL<sup>5</sup>:

```
AG (landingButi.pressed
    -> AF (lift.floor=i & lift.door=open))
```

These are some properties that we verified for the base lift system and for its extensions with features. The results of the verifications are summarised in Table 2.1. (The numbers in the table refer to the numbering in this list.)

1. Pressing a landing button guarantees that the lift will arrive at that landing and open its doors:

```
AG (landingButi.pressed
    -> AF (lift.floor=i & lift.door=open))
```

2. If a button inside the lift is pressed, the lift will eventually arrive at the corresponding floor.

```
AG (liftButi.pressed -> AF (floor=i & door=open))
```

3. The lift will not change its direction while there are calls in the direction it is travelling.

One formula for upwards travel,

```
AG  $\forall i < j.$  (floor=i & liftButj.pressed & direction=up
    -> A[direction=up U floor=j])
```

...and one formula for downwards travel:

---

<sup>5</sup> To enhance the readability of the specifications we present them in a meta-notation, using variables and quantifiers which SMV does not allow. Translating this into pure SMV notation is purely mechanical, though. In these examples, any free variables are universally quantified. For example, if we expand the above formula to pure SMV, we obtain the conjunction of the formulas:

```
AG (landingBut1.pressed -> AF (lift.floor=1 & lift.door=open))
```

through

```
AG (landingBut5.pressed -> AF (lift.floor=5 & lift.door=open))
```

---


$$\text{AG } \forall i > j. (\text{floor}=i \ \& \ \text{liftBut}j.\text{pressed} \ \& \ \text{direction}=\text{down} \\ \rightarrow \text{A}[\text{direction}=\text{up} \ \text{U} \ \text{floor}=j])$$

4. If the door closes, it may remain closed.

$$\text{!AG } (\text{door}=\text{closed} \ \rightarrow \ \text{AF } \text{door}=\text{open})$$

5. The lift may remain idle with its doors closed at floor  $i$ .

$$\text{EF } (\text{floor}=i \ \& \ \text{door}=\text{closed} \ \& \ \text{idle}) \\ \text{AG } (\text{floor}=i \ \& \ \text{door}=\text{closed} \ \& \ \text{idle} \\ \rightarrow \ \text{EG } (\text{floor}=i \ \& \ \text{door}=\text{closed}))$$

(The first formula states that the lift can actually get into a state satisfying the premise of the second formula.)

6. The lift may stop at floors 2, 3, and 4 for landing calls when travelling upwards:

$$\forall i \in \{2, 3, 4\}. \ \text{!AG } ((\text{floor}=i \ \& \ \text{!liftBut}i.\text{pressed} \\ \& \ \text{direction}=\text{up}) \ \rightarrow \ \text{door}=\text{closed})$$

7. The lift may stop at floors 2, 3, and 4 for landing calls when travelling downwards:

$$\forall i \in \{2, 3, 4\}. \ \text{!AG } ((\text{floor}=i \ \& \ \text{!liftBut}i.\text{pressed} \\ \& \ \text{direction}=\text{down}) \ \rightarrow \ \text{door}=\text{closed})$$

One can think of many more properties to check for a lift system. In this chapter, I omit all safety properties, and concentrate on a selection of properties that are characteristic of the SBCC algorithm, namely ones that concern guarantee of service (or absence thereof).

### Features of the lift system

The following features of the lift system were described using the feature construct, and then integrated into the base system using the feature integrator:

**Parking.** When a lift is idle, it goes to a specified floor (typically the ground floor) and opens its doors. This is because the next request is anticipated to be at the specified floor. In a real lift system the parking floor may be different at different times of the day, for example anticipating

## 2.4. Case studies

---

upwards-travelling passengers in the morning and downwards-travelling passengers in the evening.

**Lift- $\frac{2}{3}$ -full.** When the lift detects that it is more than two-thirds full, it does not stop in response to landing calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the lift, as serving them will help reduce its load.

**Overloaded.** When the lift is overloaded, the doors will not close. Some passengers must get out.

**Empty.** When the lift is empty, it cancels any calls which have been made inside the lift. Such calls were made by passengers who changed their mind and exited the lift early, or by practical jokers who pressed lots of buttons and then got out.

**Executive Floor.** The lift gives priority to calls from the executive floor.

By way of illustration, I give the code for the parking feature in Figure 2.7. The parking feature introduces the specification ((12) in Table 2.1)

$$\text{AG } \forall i \neq 1. \text{ !EG}(\text{floor}=i \ \& \ \text{door}=\text{closed})$$

which says that the lift will not remain idle indefinitely at any floor other than floor 1. (Figure 2.7 contains only the instance for  $i = 3$ .)

The other features introduce other specifications; these are listed below.

### Properties for the featured lift system.

In addition to the generic properties for the base system, we check some requirements for each feature. The results for these properties can also be found in Table 2.1. (Again the numbering in the table corresponds to the numbers in this list.)

I derived these requirements from the (natural language) description of the features and translated them into CTL as directly as possible.

#### 8. Empty:

The lift will not arrive empty at a floor unless the button on that landing was pressed.

$$\text{AG } (\text{lift.floor}=i \ \& \ \text{lift.door}=\text{open} \ \& \ \text{lift.empty} \\ \rightarrow \ \text{landingBut}i.\text{pressed})$$

```
FEATURE park
REQUIRE
  MODULE main -- require all landing buttons
  VAR
    landingBut1.pressed : boolean; landingBut2.pressed : boolean;
    landingBut3.pressed : boolean; landingBut4.pressed : boolean;
    landingBut5.pressed : boolean;
  MODULE lift -- require all lift buttons and the variable floor
  VAR
    floor          : {1,2,3,4,5};
    liftBut1.pressed : boolean; liftBut2.pressed : boolean;
    liftBut3.pressed : boolean; liftBut4.pressed : boolean;
    liftBut5.pressed : boolean;

INTRODUCE
  MODULE lift -- no new variables introduced
  SPEC -- lift parks at floor 1:
    AG (floor=4 & idle -> E [idle U floor=1])
  SPEC -- lift cannot park at floor 3:
    AG (!EG(floor=3 & door=closed))

CHANGE
  MODULE main
  IF !lift.floor=1 &
    !( landingBut1.pressed | lift.liftBut1.pressed |
      landingBut2.pressed | lift.liftBut2.pressed |
      landingBut3.pressed | lift.liftBut3.pressed |
      landingBut4.pressed | lift.liftBut4.pressed |
      landingBut5.pressed | lift.liftBut5.pressed )
  THEN TREAT landingBut1.pressed = 1
END
```

Figure 2.7: The code for the Parking feature



### Feature interactions in the lift system

Feature integration provides a framework for plugging these different features into the lift system, and by examining the result, witnessing feature interactions. The SFI tool integrates one or more of the features, in a given order, into the base system. The result of our experimentation with the features for the lift system is summarised in Table 2.1.

Each row represents a combination of the base system and some features, and each column represents a property which SMV has checked against the relevant systems. The first row is the unfeatured lift system; rows 2–6 represent the base system with just one feature, and the remaining rows represent the base system with two features. The order in which two features are added matters in general. In those cases where exactly the same specifications are satisfied, we write  $F_1 * F_2$  and list just one ordering, otherwise we write  $F_1 + F_2$ . (Thus, inspection of the table reveals that the only features which do not commute are Lift- $\frac{2}{3}$ -full and Executive Floor: a type IV interaction.)

The properties, represented by columns in the table, are divided into two groups. Properties 1–7, to the left of the double line, are properties which apply to any lift system, featured or not. We can see which properties are broken by the addition of various features.

To the right of the double line are properties 8–14 which are designed to test the integration of specific features. Whenever there is a cross in the right-hand part of the table, we have detected some kind of feature interference. A requirement of one of the features is not satisfied in the presence of the other feature. This initial diagnosis has to be followed by a closer look at the features and the property concerned to find out the reasons (and the seriousness) of the interference.

We can see that most feature interferences are of type I or II (*cf.* page 27), respectively, depending on the order of integration. Only combination Lift- $\frac{2}{3}$ -full + Executive Floor produces a type III interaction. As mentioned above these features also exhibit a type IV interaction.

For example, in the line “Overloaded + Empty” we can see that one of the violated properties (9) is about guaranteed service for the lift with the “Empty” feature. Obviously this cannot be expected to hold for an overloaded lift since we already know that the “Overloaded” feature can block the lift. (Service will still be guaranteed as long as `overload` is not true, but we omitted this property from the table.)

The reason for the second violation of a property of “Empty” by integrating “Overloaded” is quite different. Here the violation stems as much from the way the property was coded in CTL,

## 2.4. Case studies

Table 2.1: Feature interactions for the lift system

Feature(s)	Property (see section 2.4.1, p. 32f and 34f)													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
no features	√	√	√	√	√	√	√	—	—	—	—	—	—	—
Empty	√	×	×	√	√	√	√	√	√	—	—	—	—	—
Overloaded	×	×	×	√	√	√	√	—	—	√	√	—	—	—
Parking	√	√	√	√	×	√	√	—	—	—	—	√	—	—
Lift- $\frac{2}{3}$ -full	×	√	√	√	√	√	√	—	—	—	—	—	√	—
Exec. Floor	×	×	√	√	√	√	√	—	—	—	—	—	—	√
Overloaded * Empty	×	×	×	√	√	√	√	×	×	√	√	—	—	—
Parking * Empty	√	×	×	√	×	√	√	√	√	—	—	√	—	—
Lift- $\frac{2}{3}$ -full * Empty	×	×	×	√	√	√	√	√	√	—	—	—	×	—
Exec. Floor * Empty	×	×	×	√	√	√	√	√	×	—	—	—	—	√
Parking * Overloaded	×	×	×	√	×	√	√	—	—	√	√	√	—	—
Lift- $\frac{2}{3}$ -full * Overloaded	×	×	×	√	√	√	√	—	—	√	√	—	×	—
Exec. Floor * Overloaded	×	×	×	√	√	√	√	—	—	√	√	—	—	×
Lift- $\frac{2}{3}$ -full * Parking	×	√	√	√	×	√	√	—	—	—	—	√	√	—
Exec. Floor * Parking	×	×	√	√	√	√	√	—	—	—	—	√	—	√
Exec. Floor + Lift- $\frac{2}{3}$ -full	×	×	√	√	√	√	√	—	—	—	—	—	√	×
Lift- $\frac{2}{3}$ -full + Exec. Floor	×	×	√	√	×	√	√	—	—	—	—	—	×	×

√: property holds; ×: property does not hold; —: property not applicable

```
-- specification AG (lift.floor = 4 & lift.door = open & ... is false
-- as demonstrated by the following execution sequence
state 32.1:
landing_call = 0
no_call = 1
landingBut1.pressed = 0
landingBut2.pressed = 0
landingBut3.pressed = 0
landingBut4.pressed = 0
landingBut5.pressed = 0
lift.idle = 1
lift.car_call = 0
lift.overload = 0
lift.empty = 0
lift.floor = 1
lift.door = closed
lift.direction = down
lift.liftBut5.pressed = 0
lift.liftBut4.pressed = 0
lift.liftBut3.pressed = 0
lift.liftBut2.pressed = 0
lift.liftBut1.pressed = 0

state 32.2:
lift.idle = 0
lift.liftBut2.pressed = 1

state 32.3:
lift.car_call = 2
lift.direction = up

state 32.4:
lift.floor = 2
lift.door = open

state 32.5:
lift.idle = 1
lift.car_call = 0
lift.overload = 1
lift.empty = 1
lift.liftBut2.pressed = 0
```

Figure 2.8: An interaction between the Empty and Overloaded features

```
AG (lift.floor=i & lift.door=open & lift.empty
    -> landingButi.pressed)
```

as from the way the system and the features were coded. Essentially, in the base system, the lift would never stay at the same floor with its doors open for more than one step; and the buttons are reset in the step after the doors were opened. (*Cf.* property 5 in section 2.4.1.) Figure 2.8 shows the trace<sup>6</sup> that SMV produced for  $i = 2$ . We can see that the lift starts from its initial state of sitting idle on the ground floor (`lift.floor=1`). It is then sent to floor 2 by the button inside the lift (`lift.liftBut2.pressed=1`). It travels there and opens its doors (state 32.4). The last state violates the property: the lift button is reset, the landing button was never pressed. This is due to the Overloaded feature forcing the lift doors open as long as the lift is overloaded.

We see that the violation occurs when both the flag `overload` and `empty` are true. It is not hard to see that an interference is inevitable when both `overload` and `empty` are true. In reality a lift can never be overloaded and empty at the same time, but our verification software and the feature integrator cannot know that. One possible solution would be to alter the features to take account of this constraint. This would run counter to one of the basic ideas of features: that they are specified independently of one another and without knowledge about other features. A better solution would be to design another feature that implements the constraint by either setting `overload` to false when `empty` is true, or vice versa.<sup>7</sup>

### 2.4.2 A telephone system with eight features

The second case study for the SMV feature construct is a simple version of the Plain Old Telephone System (POTS). Features we have modelled for integration into our model of POTS include:

**Call Waiting (CW)** When the subscriber is engaged in a call, and there is a second incoming call, the subscriber is notified and the second call is put on hold. The subscriber can switch between the two calls at will. A caller will hear an announcement while her call is on hold.

**Call Forward Unconditional (CFU)** All calls to the subscriber's phone are diverted to another phone.

---

<sup>6</sup>The trace records all variables for the initial state, and after that only those that change from the previous state.

<sup>7</sup>It has been remarked that one could also introduce a variable for the load in the lift, and then use this in the definition of the features Empty, Overloaded and Lift- $\frac{2}{3}$ -full. However, then we would be talking about a different base system.

**Call Forward on Busy (CFB)** All calls to the subscriber's phone are diverted to another phone, if and when the subscriber's line is busy.

**Call Forward on No Reply (CFNR)** All calls to the subscriber's phone which are not answered after a certain amount of time, are diverted to another phone.

**Ring Back When Free (RBWF)** If the user gets the busy-tone on calling another line, she can choose to activate RBWF, which will attempt to establish a connection with that line as soon as it becomes idle.

**Terminating Call Screening (TCS)** This feature prevents calls to the subscriber's phone from any number on the screening list chosen by the subscriber. The caller will hear an announcement to the effect that her call is being rejected.

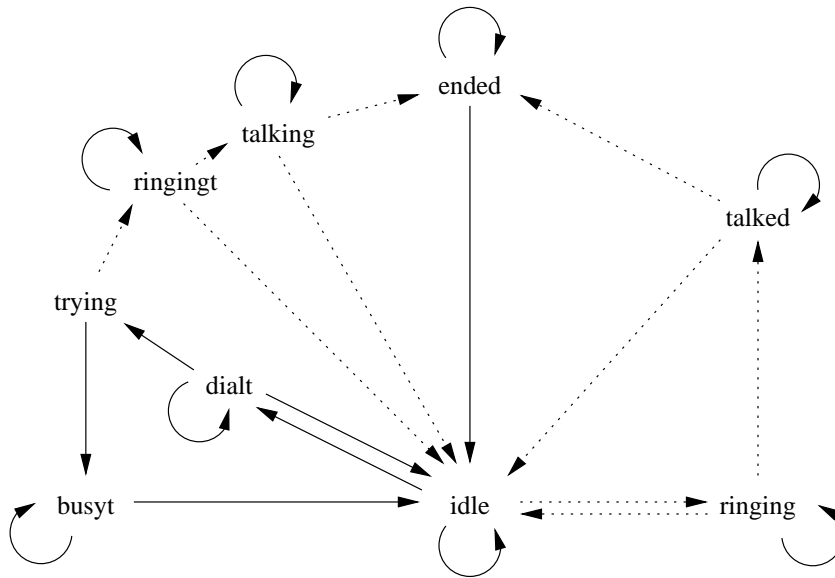
**Originating Call Screening (OCS)** This feature inhibits calls from the subscriber's phone to any number from a set chosen by the subscriber. Any attempt to ring such a number will yield an announcement.

**Automatic Call Back (ACB)** This feature records the number of the last caller to the subscriber's phone, which the subscriber can choose to ring directly, without dialling the number.

### The base system (POTS)

Mark Ryan and I have built an SMV description of a network of four synchronous telephones. The behaviour of each phone is given by the finite automaton shown in Figure 2.9, plus one variable, `dialled`, for each phone which indicates the phone to which it is connected (or to which it is trying to connect). Initially the phone is in state `idle`; from there, it may move to `ringing` (if someone rings it) or to `dialt` (if someone lifts the handset). `Dialt`, `ringingt`, and `busyt` abbreviate dial-tone, ringing-tone, and busy-tone, respectively. `Talking` represents the state where the phone is connected in a conversation which it initiated, while `talked` means that the conversation was initiated by someone else. `Ended` means that the party to which the phone was connected has hung up but the handset is still off the hook.

User input is simulated by non-determinism: the number to be dialled is non-deterministically chosen, and when there is more than one transition from a state, one is chosen non-deterministically. A number of transitions of the telephone automaton have to synchronise with transitions in another phone. The variable `dialled` determines the other copy of the phone automaton with which these transitions have to synchronise. If a transition



(Dotted lines indicate synchronising transitions.)

Figure 2.9: The automaton for a single phone.

has to synchronise with a transition in another phone (indicated by a dotted line in the diagram), it can only be chosen if the other phone chooses the corresponding transition. In detail, the transitions are synchronised as follows:

trying → ringingt	with	idle → ringing
ringingt → idle	with	ringing → idle
ringingt → talking	with	ringing → talked
talking → idle	with	talked → ended
talking → ended	with	talking → idle.

The code for the phone module can be seen in Figures 2.10 and 2.11. In this piece of code one can also see how the synchronisation mechanism helps to avoid the race condition arising when several phones try to contact the same line at the same time. In SMV we do this by using the `next` operator on the right hand side of an assignment. This way we can look at the current *and* the next state of a phone, i.e. the transition. Making assignments conditional on both current and next states of other telephone automata we can implement the above synchronisations. Details can be seen in the source code in Figures 2.10 and 2.11.

The fairness constraints at the end of the module `phone` ensure that a telephone cannot remain in the same state indefinitely. Since `st` must be

different from e.g. `idle` infinitely often along any execution, it can only be equal to `idle` finitely many steps at a time. Note that the telephone may still visit the `idle` state infinitely often.

As it turned out, this model quickly grew too large to verify when features were added, since every phone was extended with the features. Therefore we proceeded to a reduced model with only two complete phones, and one terminating and one originating phone (thus, still four in total). In the diagram (Figure 2.9), the left hand side represents the originating line, and the right hand side the terminating line, both including the states `idle` and `ended`. Additionally, each feature was only added to one of the phones (unless explicitly stated). A positive side-effect of this differentiation is that one can distinguish the interactions according to how features are distributed over the system.

For the features we modelled, we argue that the reduced model still exhibits all possible interactions of two feature instances if we go through all relevant combinations.

First we argue that four (complete) phones are sufficient. Each feature deals with at most three parties, and each phone can only originate one call (we did not model Three Way Calling). Therefore a second feature may be added on any type of phone: one that is affected by the first feature in some way, or one that is not connected to it in any way. This gives rise to all interesting behaviours.

The main premise for our reasoning is that the effects of a feature are localised, i.e. only those phones which participate in a call affected by an instance of the feature, exhibit altered behaviour. I will use the term *configuration* to describe such a set of phones. Parties outside a configuration are not affected by the feature and behave independently.

The idea of a call and its participants is central to the concept of configurations: a basic call has two participants, this we can consider a configuration. Any party outside this configuration will (by default) be turned away by a busy signal when attempting to ring either participant. Expanding this idea to a featured system, we can define a configuration as a set of phones connected in some way, e.g. talking to each other, or otherwise depending on each other in their behaviour. Starting with the featured telephone, we arrive at a configuration by adding successively any phone which displays behaviour not already observable in the configuration.

If the transitions of a telephone is changed by a feature, this is an indication only of the *potential* for changed behaviour: the states from which the changed (new or removed) transitions originate may not be satisfiable in the context of the changes to the rest of the system.

As an example, we look at the Call Waiting feature. Apart from the

## 2.4. Case studies

---

```
MODULE phone (X,B,C,D,p) -- parameters: the 4 numbers, and the array of phones
--                               X is our own number

VAR
  dialled : {0,1,2,3,4};
  st      : {idle,dialt,trying,busyt,ringingt,talking,ringing,talked,ended};

ASSIGN
  init(dialled) := 0;
  next(dialled) := case
    next(st=idle)           : 0;
    dialled = 0 & next(st)=trying : {1,2,3,4};
    1                       : dialled;
  esac;

  init(st) := idle;
  next(st) := case
    st=idle :
      case
        p[B].st=trying & p[B].dialled=X & next(p[B].st=ringingt) : ringing;
        p[C].st=trying & p[C].dialled=X & next(p[C].st=ringingt) : ringing;
        p[D].st=trying & p[D].dialled=X & next(p[D].st=ringingt) : ringing;
        1                                     : {idle,dialt};
      esac;
    st=ringing :
      case
        p[B].st=ringingt & p[B].dialled=X & next(p[B].st)=idle : idle;
        p[C].st=ringingt & p[C].dialled=X & next(p[C].st)=idle : idle;
        p[D].st=ringingt & p[D].dialled=X & next(p[D].st)=idle : idle;
        1                                     : {ringing,talked};
      esac;
    st=dialt : {dialt,trying};
    st=busyt : {idle,busyt};
    st=trying :
      case
        dialled=B & p[B].st=idle
          & ((p[C].st=trying & p[C].dialled=B)->next(p[C].st)=busyt)
          & ((p[D].st=trying & p[D].dialled=B)->next(p[D].st)=busyt) : ringingt;
        dialled=C & p[C].st=idle
          & ((p[B].st=trying & p[B].dialled=C)->next(p[B].st)=busyt)
          & ((p[D].st=trying & p[D].dialled=C)->next(p[D].st)=busyt) : ringingt;
        dialled=D & p[D].st=idle
          & ((p[B].st=trying & p[B].dialled=D)->next(p[B].st)=busyt)
          & ((p[C].st=trying & p[C].dialled=D)->next(p[C].st)=busyt) : ringingt;
        1                                     : busyt;
      esac;
    st=ringingt :
      case
        dialled=B & next(p[B].st)=talked : talking;
        dialled=C & next(p[C].st)=talked : talking;
        dialled=D & next(p[D].st)=talked : talking;
        1                                     : {ringingt,idle};
      esac;
```

Figure 2.10: The SMV code for the phone system. (1/2)

```

st=talked :
  case
    p[B].st=talking & p[B].dialled=X & next(p[B].st)=idle : ended;
    p[C].st=talking & p[C].dialled=X & next(p[C].st)=idle : ended;
    p[D].st=talking & p[D].dialled=X & next(p[D].st)=idle : ended;
    1 : {idle,talked};
  esac;

st=talking :
  case
    dialled=B & p[B].st=talked & next(p[B].st)=idle : ended;
    dialled=C & p[C].st=talked & next(p[C].st)=idle : ended;
    dialled=D & p[D].st=talked & next(p[D].st)=idle : ended;
    1 : {idle,talking};
  esac;

st=ended : {ended,idle};
esac;

-- Fairness constraints to ensure that a phone does not remain in a state
-- indefinitely. A phone may still alternate between, eg, idle and dialt.
FAIRNESS !st=idle
FAIRNESS !st=dialt
FAIRNESS !st=trying
FAIRNESS !st=busyt
FAIRNESS !st=ringingt
FAIRNESS !st=talking
FAIRNESS !st=ringing
FAIRNESS !st=talked
FAIRNESS !st=ended

MODULE main
VAR
  ph[1] : phone (1,2,3,4,ph);
  ph[2] : phone (2,1,3,4,ph);
  ph[3] : phone (3,1,2,4,ph);
  ph[4] : phone (4,1,2,3,ph);

```

Figure 2.11: The SMV code for the phone system. (2/2)

subscriber’s phone we need to have two other (unfeatured) phones in a configuration: the first one can connect to the subscriber’s phone (this requires at least two phones), the second because it can be put on hold when connecting to the subscriber’s phone, which could only happen with the subscriber already involved in a call to another party, i.e. with the two previous phones in the configuration.

In this sense Call Forwarding features, too, should require a three-phone configuration, the subscriber, the caller and the target of the diversion. However, in our implementation, Call Forwarding changes the behaviour only of the calling telephone: instead of connecting to the original target of the call, the dialled number is changed – after this the call proceeds as if this number had been dialled initially. The subscriber’s phone does not display new behaviour, since it never ‘notices’ that someone attempted to call it; similarly the target of the diverted call sees only a normal incoming call.

From this example it is clear that we can sometimes find smaller sufficient configurations than those produced by the rules above if we use more detailed knowledge about the implementation. I will use this fact in the argument that follows.

The procedure for finding all configurations for the feature could be mechanised, however with considerable effort, unless the method for describing features is already geared towards this end. This is the case for the topological and combinatorial approaches mentioned on page 1.5.1. For the SMV case study I did not go to such lengths.

With this notion of configuration, we can now assume that phones outside a configuration do not exhibit new behaviour (with regard to the feature giving rise to that configuration). A direct consequence of this assumption is that we only need to look at overlapping configurations when checking combinations of features. In essence, we are taking some of the choice away from the model checker, and in turn we need to ensure that we test all relevant cases. Checking different ways of overlapping will in general happen through the model checking process.

For the features we modelled in our case study, configurations comprise one, two or three phones. Only Call Waiting affects three phones, since the Call Forwarding features operate by re-routing the call (see above). From this it is clear that overlapping configurations in a system with only two feature instances (from the set of features we consider) contain at most four distinct phones.

The argument for the soundness of the abstraction is more difficult, and depends on the fact that any phone can only originate one call, among others. Here we have to look at the particular features in more detail and determine what “roles” the phones in a configuration can take.

To illustrate this type of reasoning we look at Call Waiting. Call Waiting has up to three distinct “roles”: the subscriber, the party that the subscriber called, and a (subsequent) caller to the subscriber.<sup>8</sup> Obviously, the non-subscriber roles can be filled by the truncated phones, so that we still have a full phone to apply any other feature to. (Of course, this phone may also become part of a Call Waiting configuration.) This phone with its feature instance may now exercise the behaviour with respect to all possible roles in the Call Waiting configuration. Similar arguments apply for other features; however, they are simpler since Call Waiting is the only feature in the case study that affects up to three phones at once.

### Integrating features into the telephone system

As an illustration of the feature construct I show the Ring Back When Free feature in Figure 2.12. When looking at this example the reader should keep in mind that this code was written with the goal to run it through a model checker – and that the syntax which SMV accepts is rather limited. So for efficiency reasons, RBWF will only store one number at a time, and we do not allow cancelling RBWF once it is activated, until a call between the subscribed phone and the phone with the stored number has been established.

The **REQUIRE** section states that the feature needs a **MODULE phone** with at least the named parameters, and within that module, variables **dialed** and **st** are required, and the domain of **dialed** has to include at least the values 0 through 4, and that of **st** the values **idle**, **trying**, **busyt**, **talking** and **talked**.

The code given in the **INTRODUCE** section declares two new variables, **rbwf-use** and **rbwf-number**, and defines which number to store in **rbwf-number**, and under which conditions RBWF may be activated (**rbwf-use=1**) and deactivated (**rbwf-use=0**).

Finally, in the **CHANGE** section we define how the new variables interact with those of the base system. For the RBWF feature, the **CHANGE** section states that when both the subscriber’s phone and the phone whose number was stored are idle, the subscriber’s phone should try to connect to the phone whose number was stored.

We did not model the subscriber’s phone ringing to alert her to the fact that the RBWF call is being attempted, although this would not be difficult; in fact, this could even be implemented as another feature. It would, however,

---

<sup>8</sup>If both calls are incoming calls, the situation is symmetrical and there are only two distinct roles.

## 2.4. Case studies

---

```
FEATURE rbwf -- Ring Back When Free
REQUIRE
  MODULE phone(X,B,C,D,p) -- req'd parameters: our number and those of the
  VAR -- other phones, and the array of phones
    dialled : {0,1,2,3,4};
    st      : {idle,trying,busy,talking,talked};

INTRODUCE
  MODULE phone
  VAR
    rbwf-number : {0,1,2,3,4}; -- to store the number we're trying to reach
    rbwf-use     : boolean;    -- true if RBWF activated
  ASSIGN
    init(rbwf-number) := 0;
    next(rbwf-number) :=
      case
        rbwf-number=0 -- don't allow changing the stored number
          & st=busy & rbwf-use : dialled;
        !rbwf-use : 0 -- reset stored number on deactivation
        1 : rbwf-number;
      esac;
    init(rbwf-use) := 0;
    next(rbwf-use) :=
      case
        rbwf-use -- only deactivate if call established (either way)
          &((dialled=rbwf-number & st=talking)
            |(st=talked
              &(rbwf-number=B & p[B].st=talking & p[B].dialled=X)
              &(rbwf-number=C & p[C].st=talking & p[C].dialled=X)
              &(rbwf-number=D & p[D].st=talking & p[D].dialled=X))) : 0;
        !rbwf-use & st=busy : {0,1}; -- may activate RBWF on busy-tone
        1 : rbwf-use; -- otherwise, keep same value
      esac;

CHANGE
  MODULE phone
    IF (rbwf-use & st=idle -- if RBWF is active and our phone is idle
      &((rbwf-number=B & p[B].st=idle) -- and the stored phone is idle,
        |(rbwf-number=C & p[C].st=idle) -- try to connect to it
        |(rbwf-number=D & p[D].st=idle)))
    THEN IMPOSE next(dialled) := rbwf-number;
      next(st) := trying;

END
```

Figure 2.12: The code for the feature Ring Back When Free

further increase the size of the model, as we would have to introduce a new variable to indicate the special ringing.

Apart from the generic properties of the phone system listed in the next section (2.4.2), we also want to verify that the base system with the feature actually behaves as the feature specification demands. For example, in the case of RBWF we also require the following (omitted in Figure 2.12):

- If RBWF is active, the stored number will be dialled as soon as possible (as long as RBWF is active).

$$\begin{aligned} \text{AG } \forall i \neq j. & ((\text{ph}[i].\text{rbwf-use} \ \& \ \text{ph}[i].\text{rbwf-number}=j) \\ & \rightarrow \text{A} [ (\text{ph}[i].\text{st=idle} \ \& \ \text{ph}[j].\text{st=idle} \\ & \quad \rightarrow \text{AX } \text{ph}[i].\text{dialled}=j) \\ & \quad \text{W } !\text{ph}[i].\text{rbwf-use} \ ] ) \end{aligned}$$

- The stored number is reset when a call to the stored number is completed.

$$\begin{aligned} \text{AG } \forall i \neq j. & ((\text{ph}[i].\text{rbwf-number}=j \\ & \quad \& \ \text{ph}[i].\text{st=talking} \ \& \ \text{ph}[i].\text{dialled}=j) \\ & \rightarrow \text{AF } \text{ph}[i].\text{rbwf-number}=0) \end{aligned}$$

The stored number is also reset when the target party calls.

$$\begin{aligned} \text{AG } \forall i \neq j. & ((\text{ph}[i].\text{rbwf-number}=j \ \& \ \text{ph}[i].\text{st=talked} \\ & \quad \& \ \text{ph}[j].\text{dialled}=i \ \& \ \text{ph}[j].\text{st=talking}) \\ & \rightarrow \text{AF } \text{ph}[i].\text{rbwf-number}=0) \end{aligned}$$

- RBWF is deactivated when a call to the stored number is completed.

$$\begin{aligned} \text{AG } \forall i \neq j. & ((\text{ph}[i].\text{rbwf-number}=j \\ & \quad \& \ \text{ph}[i].\text{st=talking} \ \& \ \text{ph}[i].\text{dialled}=j) \\ & \rightarrow \text{AF } \text{ph}[i].\text{rbwf-use}=0) \end{aligned}$$

RBWF is also deactivated when the target party calls.

$$\begin{aligned} \text{AG } \forall i \neq j. & ((\text{ph}[i].\text{rbwf-number}=j \ \& \ \text{ph}[i].\text{st=talked} \\ & \quad \& \ \text{ph}[j].\text{dialled}=i \ \& \ \text{ph}[j].\text{st=talking}) \\ & \rightarrow \text{AF } \text{ph}[i].\text{rbwf-use}=0) \end{aligned}$$

Note that we can also verify the converse: that RBWF is deactivated, or that the number reset, only when intended, i.e. when a call to the number is completed: (using the shorthand ‘weak until’ defined on page 11)

$$\text{AG } \forall i \neq j. (\text{ph}[i].\text{rbwf-use} \rightarrow \\ \text{A} [ \text{ph}[i].\text{rbwf-use} \\ \text{W } \text{ph}[j].\text{dialled}=i \ \& \ \text{ph}[j].\text{st=talking} ])$$

As expected the base system plus the Ring Back When Free feature satisfies these specifications. After all, these were the requirements for the feature. We also found that RBWF does not violate any of the properties that we stipulated for the base system. (See Table 2.3, and the following section.)

### Properties of the basic phone system.

These are the properties that we have verified for the base system. (Again we use the meta-notation introduced on page 32.)

To save space, I have omitted from Table 2.3 some more technical properties, but also these rather intuitive properties of the base system:

- The correct phone will ring: if phone  $i$  is trying to contact phone  $j$  and consequently gets the ringing-tone, then phone  $j$  must be ringing.

$$\text{AG } ((\text{ph}[i].\text{st=trying} \ \& \ \text{ph}[i].\text{dialled}=j) \\ \rightarrow \text{AX } (\text{ph}[i].\text{st=ringingt} \rightarrow \text{ph}[j].\text{st=ringing}))$$

- Phone  $i$  can be talked to; and if it is being talked to, there has to be another phone talking to it.

$$\text{EF } \text{ph}[i].\text{st=talked} \\ \text{AG } (\text{ph}[i].\text{st=talked} \\ \leftrightarrow \exists j. (\text{ph}[j].\text{st=talking} \ \& \ \text{ph}[j].\text{dialled}=i))$$

- Phone  $i$  may ring, and if it is ringing there has to be another phone that has dialled it and is getting the ringing-tone.

$$\text{EF } \text{ph}[i].\text{st=ringing} \\ \text{AG } (\text{ph}[i].\text{st=ringing} \\ \leftrightarrow \exists j. (\text{ph}[j].\text{st=ringingt} \ \& \ \text{ph}[j].\text{dialled}=i))$$

The results for the following properties are given in Table 2.3:

1. Any phone may call any other phone.

$$\text{AG } \forall i \neq j. \text{EF } (\text{ph}[i].\text{st=talking} \ \& \ \text{ph}[i].\text{dialled}=j)$$

2. If phone  $i$  is talking to phone  $j$ , the call will eventually end; and this will be by one party hanging up ( $\text{st=idle}$ ) and the other party still off-hook ( $\text{st=ended}$ ). (This holds only with “weak” fairness, which ensures that a phone cannot remain in the same state indefinitely.)

$$\text{AG } ((\text{ph}[i].\text{dialled}=j \ \& \ \text{ph}[i].\text{st}=\text{talking}) \\ \rightarrow \text{AF } ((\text{ph}[i].\text{st}=\text{idle} \ \& \ \text{ph}[j].\text{st}=\text{ended}) \ | \\ (\text{ph}[j].\text{st}=\text{idle} \ \& \ \text{ph}[i].\text{st}=\text{ended})))$$

3. When a phone is in state `trying`, it will always get ringing-tone or busy-tone in the next step.

$$\text{AG } (\text{ph}[i].\text{st}=\text{trying} \\ \rightarrow \text{AX } (\text{ph}[i].\text{st}=\text{ringingt} \ | \ \text{ph}[i].\text{st}=\text{busyt}))$$

4. If a phone is talking, the dialled phone must be talked to.

$$\text{AG } (\text{ph}[i].\text{st}=\text{talking} \ \& \ \text{ph}[i].\text{dialled}=j \\ \rightarrow \text{ph}[j].\text{st}=\text{talked})$$

5. Never can two phones be talking to the same third phone.

$$\text{AG } \forall i \neq j. \ !(\text{ph}[i].\text{st}=\text{talking} \ \& \ \text{ph}[i].\text{dialled}=k \ \& \\ \text{ph}[j].\text{st}=\text{talking} \ \& \ \text{ph}[j].\text{dialled}=k \ )$$

6. The dialled number cannot change without replacing the hand-set. (This only holds with “weak” fairness, otherwise phone  $i$  could stay in state `trying` indefinitely.<sup>9</sup>)

$$\text{AG } ((\text{ph}[i].\text{dialled}=j \ \& \ \text{ph}[i].\text{st}=\text{trying}) \\ \rightarrow (\text{A}[\ \text{ph}[i].\text{dialled}=j \ \cup \ \text{ph}[i].\text{st}=\text{idle}]))$$

### Properties for the featured phone system.

The following requirements were derived from the description of the services that these features implement and verified for the respective features. (For a description of the features see page 2.4.2.) For lack of space I only give one or two properties for each feature and omit some of the more technical properties that I verified.

<sup>9</sup>In absence of the fairness constraints, one would use the ‘weak until’ connective, *cf.* page 11.

## 2.4. Case studies

---

### 7. Call Forwarding Unconditional:

If a forwarding number is given, the phone will never receive calls. (The forwarding number is chosen at random at initialisation but does not change after that.)

```
AG (!ph[i].cfu-forw=0
  -> AG !(ph[i].st in {ringing,talked}))
```

### 8. Call Forwarding on Busy:

If the subscriber's phone is busy, incoming calls will terminate at the phone with the forwarding number. (Again, the forwarding number remains fixed.)

```
AG  $\forall i \neq j \neq k \neq i$ . ((ph[i].cfb-forw=j & !ph[i].st=idle
  & ph[k].dialled=i & ph[k].st=trying)
  -> AF(ph[k].dialled=j & ph[k].st in {busyt,ringingt}
  & (ph[k].st=ringingt -> ph[j].st=ringingt)))
```

### 9. Call Waiting:

If there are two calls to the subscribers phone, exactly one party will hear the Call Waiting message, "Your call is on hold." (In other words, at most one party will hear the message at any given time.)

```
AG  $\forall i \neq j \neq k \neq i$ . (ph[i].st=talking & ph[i].dialled=k &
  ph[j].st=talking & ph[j].dialled=k
  -> (ph[i].cw-msg <-> !ph[j].cw-msg))
```

### 10. Call Waiting:

The active party is never on hold. (In the Call Waiting feature, dialled holds the value of the party to whom the subscriber is currently talking.)

```
AG (!ph[i].dialled=0 -> !ph[i].onhold=ph[i].dialled)
```

### 11. Ring Back When Free:

If Ring Back When Free is activated, call completion will be attempted when possible, i.e. whenever both phones are idle.

```
AG  $\forall i \neq j$  ((ph[i].rbwf-use & ph[i].rbwf-number=j)
  -> A[ (ph[i].st=idle & ph[j].st=idle
  -> AX ph[i].dialled=j)
  W !ph[i].rbwf-use ])
```

## 12. Ring Back When Free:

The stored number will be reset when a call between the subscriber and the phone with the stored number is established. One formula for calls initiated by the subscriber and one for incoming calls. (These two could be rolled into one.)

```
AG  $\forall i \neq j$  ((ph[i].rbwf-number=j & ph[i].st=talking
    & ph[i].dialled=j) -> AF ph[i].rbwf-number=0)
AG  $\forall i \neq j$  ((ph[i].rbwf-number=j & ph[i].st=talked
    & ph[j].dialled=i & ph[j].st=talking
    -> AF ph[i].rbwf-number=0)
```

## 13. Terminating Call Screening:

Calls from numbers on the screening list (the array `tcs`) are never accepted.

```
AG (ph[i].tcs[j]
    -> AG !(ph[j].dialled=i
    & ph[j].st in {ringingt,talked}))
```

## 14. Originating Call Screening:

Calls to numbers on the screening list (the array `ocs`) never succeed.

```
AG (ph[i].ocs[j]
    -> AG !(ph[i].dialled=j
    & ph[i].st in {ringingt,talking}))
```

### Adding multiple features to the telephone system

So far we have only verified the correct operation of a single feature added to the base system. More interesting with view to *feature interaction* is the question if adding other features leads to violations of the specifications which the base system plus RBWF satisfies, or of specifications which are satisfied by the base system plus the respective other features.

For example, when we added RBWF to POTS+CFB, the only properties that were not preserved, were already violated by CFB on its own:

- lines calling the CFB subscriber do not have to go immediately from state `trying` to state `busyt` or `ringingt` because the diversion takes one execution step;
- the dialled number may change without replacing the hand-set when it is updated by the forwarding feature.

## 2.4. Case studies

---

The same was true when we added the features in the opposite order (first CFB, then RBWF) and irrespective of whether the same phone subscribed to both of these features or they were activated for two different phones.<sup>10</sup> This leads us to the conclusion that Call Forwarding on Busy and Ring Back When Free do not interfere with each other, at least with respect to our specification of the system. This was true regardless which telephones the features were added to.

With other features, however, RBWF is not always so well behaved. When we added RBWF to POTS+CW, we found that that RBWF did not respect the specifications introduced for CW (Type II interaction): this combination of features violated a requirement for CW (property (9) on page 51). The violated property states, that when there are two callers to a CW subscriber, exactly one of them is on hold at any given time.

```
AG (ph[2].st=talking & ph[2].dialled=1 &
    ph[3].st=talking & ph[3].dialled=1
    -> (ph[2].cw-msg <-> !ph[3].cw-msg))
```

where `ph[1]` is the phone subscribing to CW and the flag `cw-msg` indicates whether the respective phone is on hold. The trace that SMV produces as a counter-example shows up the following behaviour:

1. `ph[1]` tries to ring `ph[4]` when `ph[4]` is busy, and `ph[1]` activates RBWF;
2. `ph[1]` then calls `ph[2]` (successfully);
3. using CW, `ph[1]` accepts an incoming call from `ph[3]`, which is put on hold;
4. finally `ph[1]` hangs up on `ph[2]`, while the call from `ph[3]` is on hold and `ph[4]` is idle.
5. At this moment RBWF takes action: RBWF assumes that `ph[1]` is now idle and ready to complete the call to `ph[4]`, while, in fact, CW should let the subscriber know that she still has a call on hold.

At first sight the trace that SMV produced looked rather pathological, but that is just because a counter-example has to be a “worst case” scenario. CW may still work correctly as may be checked by

---

<sup>10</sup>The latter result was omitted from Table 2.3.

```
EG (ph[2].st=talking & ph[2].dialled=1 &
    ph[3].st=talking & ph[3].dialled=1
    -> (ph[2].cw-msg <-> !ph[3].cw-msg))
```

which turns out to be true. However, this only happens when RBWF is not activated, as can be verified by checking

```
EG ((ph[2].st=talking & ph[2].dialled=1 &
    ph[3].st=talking & ph[3].dialled=1
    -> (ph[2].cw-msg <-> !ph[3].cw-msg)) -> rbwf-use=0)
```

which also holds.

If, on the other hand, we integrate RBWF first and then CW, the system violates the RBWF requirements (Type II), namely that call completion will be attempted whenever both the subscriber's phone and the phone which RBWF should monitor become idle. This is in a sense symmetrical to the above interference, since now CW overrides RBWF when both features are activated.

Table 2.2: Interactions between features for the phone system

F <sub>1</sub>	F <sub>2</sub>						
	CW	CFU	CFB	RBWF	RBWF <sup>1</sup>	TCS	OCS
CW	—	IV	IV	II, IV	✓	IV	II <sup>4</sup>
CFU	I, IV	—	IV	✓	✓	✓	✓
CFB	I, II, IV	II, IV	—	✓	✓	II	II
RBWF	II, IV	✓	✓	—	✓	✓	✓
RBWF <sup>1</sup>	✓	✓	✓	✓	—	✓	✓
TCS	II, IV	✓	I	✓	✓	—	✓
OCS	I <sup>4</sup>	✓	I	✓	✓	✓	—

Table 2.2 indicates interferences between features for the phone system. A tick denotes that there is no interference, i.e. that both features work correctly together and it does not matter in what order they are integrated. When that is not the case, the table gives the types of interaction that we observed, according to the classification in 2.3.4. The superscripted numbers have the same meanings as in Table 2.3 and are explained below.

Table 2.3 summarises my experimental findings. Again, rows and columns represent feature combinations and properties, respectively. A ‘+’ between two features indicates that the order they are integrated into the system matters, i.e. different properties are satisfied by the two different orderings;

## 2.4. Case studies

Table 2.3: Feature interactions for the telephone system

Feature(s)	Property (see sections 2.4.2 and 2.4.2)													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
POTS	√	√	√	√	√	√	—	—	—	—	—	—	—	—
CW	√	×	×	×	×	×	—	—	√	√	—	—	—	—
CFU	×	√	×	√	√	×	√	—	—	—	—	—	—	—
CFB	√	√	×	√	√	×	—	√	—	—	—	—	—	—
RBWF	√	√	√	√	√	√	—	—	—	—	√	√	—	—
TCS	×	√	√	√	√	√	—	—	—	—	—	—	√	—
OCS	×	√	√	√	√	√	—	—	—	—	—	—	—	√
CW + CFU	×	×	×	×	×	×	√	—	√	√	—	—	—	—
CFU + CW	×	×	×	×	×	×	√	—	√	×	—	—	—	—
CW + CFB	√	×	×	√	×	×	—	√	√	√	—	—	—	—
CFB + CW	√	×	×	√	×	×	—	×	√	×	—	—	—	—
CW + RBWF	√	×	×	×	×	×	—	—	×	√	√	√	—	—
RBWF + CW	√	×	×	×	×	×	—	—	√	√	×	√	—	—
CW * RBWF <sup>1</sup>	√	×	×	×	×	×	—	—	√	√	√	√	—	—
CW + TCS	×	×	×	×	×	×	—	—	√	√	—	—	√	—
TCS + CW	√	×	×	×	×	×	—	—	√	√	—	—	×	—
CW * OCS	×	×	×	×	×	×	—	—	×	√	—	—	—	√
CFU + CFB	×	√	×	√	√	×	√	√	—	—	—	—	—	—
CFB + CFU	×	√	×	√	√	×	√	×	—	—	—	—	—	—
RBWF * CFU	×	√	×	√	√	×	√	—	—	—	√	√	—	—
TCS * CFU <sup>2</sup>	×	√	×	√	√	×	√	—	—	—	—	—	√	—
OCS * CFU <sup>2</sup>	×	√	×	√	√	×	√	—	—	—	—	—	—	√
RBWF * CFB	√	√	×	√	√	×	—	√	—	—	√	√	—	—
TCS * CFB <sup>2</sup>	×	√	×	√	√	×	—	×	—	—	—	—	√	—
OCS * CFB <sup>2</sup>	×	√	×	√	√	×	—	×	—	—	—	—	—	√
TCS * RBWF <sup>3</sup>	×	√	√	√	√	√	—	—	—	—	√	√	√	—
OCS * RBWF <sup>3</sup>	×	√	√	√	√	√	—	—	—	—	√	√	—	√

while a ‘\*’ indicates that the order does not matter. In these tables, all features are subscribed to by the same phone, unless stated otherwise (see below). The following notes interpret the superscripted numbers:

<sup>1</sup> Ring Back When Free subscribed to by a different phone.

<sup>2</sup> Call forwarding on Busy/Unconditional subscribed to by two phones.

<sup>3</sup> Call Screening subscribed to by two phones.

<sup>4</sup> This is clearly an artifact, generated by the fact that Call Waiting stores the currently active party in `dialled`, regardless whether that line is the originating or terminating line of the current connection. Hence OCS will interrupt a Call Waiting call that was established by a call *from* a phone on the screening list *to* the OCS subscriber.

It is obvious from the entries in columns 13 and 14 in Table 2.3 for the combinations of Call Forwarding and Call Screening features that the interactions we could detect among them were determined by our decision not to model Call Forwarding by routing the call through the forwarding phone. Both sides of the call ‘know’ who the other party is, thus the screening works as intended. Had we modelled the call being routed via the subscriber’s phone, the combination of a Call Forwarding feature with a Call Screening feature would have violated the Call Screening property, since the extra call leg would interfere with determining the originator and terminator of a call, which is essential for screening the call.

On a 400MHz SUN server, the model checking runs to produce these results took anything from a couple of minutes to over an hour to complete. Each run checked all the properties listed (and more, typically over 100 CTL formulas in all) for one combination of features. For most runs, memory requirements were very moderate, usually 2–10MB. It is, however, very hard to give definite values over all because the time and memory requirements vary greatly, depending on the combination of features, the properties, settings of SMV options. Even the same two features integrated in different order could lead to very different time and memory requirements in verification. Fairness often slows down SMV considerably, with weak fairness (the system cannot stay in one state indefinitely) taking less time than strong fairness (every state has to be visited infinitely often).

SMV, like any model checker, suffers from the state explosion problem. The more variables are introduced, the larger the state space becomes - and even using BDDs, in the worst case the increase in memory needed is exponential in the number of variables. For a system like the telephone system, which is highly interconnected, it is difficult to find good variable orderings in order to decrease the memory needed. So, in this case, BDDs do not save us from the state explosion problem.

## 2.5 The semantics of the SMV feature construct

Before we can delve into the formal semantics of the feature construct for SMV, we need to define the semantics of SMV. We will then derive the feature construct semantics from the syntactic manipulations that define feature integration. Finally, armed with these semantics we can prove some properties of SMV features in section 2.5.4.

For this purpose Mark Ryan and I have developed semantics for SMV which are quite different in character (more denotational) than McMillan’s as given in [55]. Our semantics make it easier to deal with the “next” operator on the right hand side of assignments, a usage which SMV supports for a large class of programs, but which McMillan does not cover in his semantics. A further extension (which is not supported by CMU SMV) is that we can allow non-deterministic expressions as conditions in “case” statements.

### 2.5.1 The syntax of SMV revisited

We assume that only one module is defined (since, anyway, in the synchronous case the SMV model checker flattens a multi-module system to a single large module). An SMV program then consists of variable declarations, “ $x : type$ ”, and assignments, “ $next(x) := e$ ” and “ $init(x) := e$ ”. The latter kind of assignment serves to define the set of initial states of the resulting automaton.

Types are (essentially) finite sets of values with certain operations on them. Expressions take the form

$$e ::= c \mid x \mid next(x) \mid e_1 \circ e_2 \mid e_1 \text{ union } e_2 \mid \begin{array}{l} \text{case} \\ ce_1 : e_1; \\ ce_2 : e_2; \\ \vdots \\ ce_n : e_n; \\ \text{esac} \end{array}$$

where  $c$  is a constant,  $x$  a variable, and  $ce_i$  is a conditional expression (i.e. an expression of boolean type). We often write  $next(e)$  for the expression  $e$  with all variables  $x_1, \dots$  replaced by  $next(x_1), \dots$

### 2.5.2 The semantics of SMV

**Definition 2.5.1** 1. Let  $P$  be an SMV text consisting of a single module. Let  $n$  be the number of variables which occur in  $P$ , and  $I = \{1, \dots, n\}$ .

We will call the  $i$ th variable  $x_i$ .

2. Every type denotes a finite set. The type of a variable  $x_i$  is written  $\text{type}(x_i)$ .
3. The set of states is the product of the domains of all state variables:  
 $S = \prod_{i \in I} \llbracket \text{type}(x_i) \rrbracket$ .
4. If  $s \in S$ , we write  $s(x_i)$  for the value of  $x_i$  in  $s$ , i.e. the  $i$ th component of  $s$ .

Let  $e$  be an expression in SMV. Its denotation  $\llbracket e \rrbracket$  is a function in  $S \times S \rightarrow \mathfrak{P}(\text{type}(e))$ . Applying  $\llbracket e \rrbracket$  to  $(s, s')$  returns the set of values that  $e$  could evaluate to if the current state is  $s$  and the next state is  $s'$ ; note that, because  $e$  may refer to next-state variables as well as current-state variables, both the current state and the next state are required to evaluate it. The result of  $\llbracket e \rrbracket(s, s')$  is a set, because the expression  $e$  is (in general) non-deterministic.

**Definition 2.5.2** The denotation of expressions is defined as follows, where  $e_1, e_2, \dots$  are expressions,  $ce_1, \dots$  are boolean expressions and  $\circ$  any binary operator (such as  $+, -, *, \&, \dots$ ):

1. If  $d$  is a constant, then  $\llbracket d \rrbracket = \lambda ss'. \{d\}$ .
2. If  $x$  is a variable, then

$$\llbracket x \rrbracket = \lambda ss'. \{s(x)\}, \text{ and } \llbracket \text{next}(x) \rrbracket = \lambda ss'. \{s'(x)\}.$$

3.  $\llbracket e_1 \circ e_2 \rrbracket = \lambda ss'. \left\{ v_1 \llbracket \circ \rrbracket v_2 \mid v_1 \in \llbracket e_1 \rrbracket(s, s'), v_2 \in \llbracket e_2 \rrbracket(s, s') \right\}$ ,  
 where  $\circ$  is one of the operations  $+, -, *, \&, \dots$ .

4.  $\llbracket e_1 \text{ union } e_2 \rrbracket = \lambda ss'. \left( \llbracket e_1 \rrbracket(s, s') \cup \llbracket e_2 \rrbracket(s, s') \right)$ ;  
 note that union in SMV denotes non-deterministic choice, and the expression  $\{1, 2, 3\}$  is just shorthand for 1 union 2 union 3.

5.  $\llbracket \text{ case}$   
 $\quad ce_1 : e_1;$   
 $\quad ce_2 : e_2;$   
 $\quad \vdots$   
 $\quad ce_n : e_n;$   
 $\text{ esac } \rrbracket$

## 2.5. The semantics of the SMV feature construct

---

$$\begin{aligned}
&= \lambda ss'. \left( \left\{ v \mid 1 \in \llbracket ce_1 \rrbracket(s, s'), v \in \llbracket e_1 \rrbracket(s, s') \right\} \right. \\
&\quad \cup \left\{ v \mid 0 \in \llbracket ce_1 \rrbracket(s, s'), 1 \in \llbracket ce_2 \rrbracket(s, s'), v \in \llbracket e_2 \rrbracket(s, s') \right\} \\
&\quad \cup \left\{ v \mid 0 \in \llbracket ce_1 \rrbracket(s, s') \cap \llbracket ce_2 \rrbracket(s, s'), \right. \\
&\quad\quad\quad \left. 1 \in \llbracket ce_3 \rrbracket(s, s'), v \in \llbracket e_3 \rrbracket(s, s') \right\} \\
&\quad \vdots \\
&\quad \left. \cup \left\{ 1 \mid 0 \in \bigcap_{i=1}^n \llbracket ce_i \rrbracket(s, s') \right\} \right)
\end{aligned}$$

Recall that in SMV, 0 denotes false and 1 denotes true; therefore,  $0 \in \llbracket ce_3 \rrbracket(s, s')$  means that  $ce_3$  can evaluate to false in  $(s, s')$ , etc. The last set in this union reflects the fact that if all the conditions  $ce_i$  evaluate to false, the case expression is defined to evaluate to 1.

When an expression  $e$  does not contain `next()`, we often write  $\llbracket e \rrbracket(s)$  rather than  $\llbracket e \rrbracket(s, s')$  to emphasise that  $\llbracket e \rrbracket$  depends only on the current state.

**Example 2.5.3** The expression

```

case
  b : a + 1;
  1 : a - 1;
esac

```

denotes

$$\begin{aligned}
&\lambda ss'. \left( \left\{ v \mid 1 \in \llbracket b \rrbracket(s, s'), v \in \llbracket a + 1 \rrbracket(s, s') \right\} \right. \\
&\quad \left. \cup \left\{ v \mid 0 \in \llbracket b \rrbracket(s, s'), 1 \in \llbracket 1 \rrbracket(s, s'), v \in \llbracket a - 1 \rrbracket(s, s') \right\} \right).
\end{aligned}$$

If  $a$  and  $b$  are variables (as opposed to other kinds of expressions), then they evaluate deterministically and we obtain

$$\lambda ss'. \begin{cases} s(a) + 1 & \text{if } s(b) \\ s(a) - 1 & \text{otherwise} \end{cases}$$

**Example 2.5.4** The expression

```

case
  case
    a = 1 : b;
    1      : !b;
  esac
  1      : a + 1;
esac
1      : case
          c : 0;
          1 : a;
        esac
esac
    
```

where  $a, b, c$  are variables, denotes

$$\lambda ss'. \begin{cases} s(a) + 1 & \text{if } (s(a) = 1 \wedge s(b)) \vee (s(a) \neq 1 \wedge \neg s(b)) \\ 0 & \text{if } (s(a) = 1 \wedge \neg s(b) \wedge s(c)) \\ & \vee (s(a) \neq 1 \wedge s(b) \wedge s(c)) \\ s(a) & \text{if } (s(a) = 1 \wedge \neg s(b) \wedge \neg s(c)) \\ & \vee (s(a) \neq 1 \wedge s(b) \wedge \neg s(c)) \end{cases}$$

The semantics of expressions are thus quite straightforward. However, they fail an important property of substitutivity. We might expect that if two expressions  $e_1, e_2$  denote the same thing, and we substitute for the variable  $x$  a third expression  $e$ , the resulting expressions should also denote the same thing. Let  $e_1[e/x]$  mean the expression  $e_1$  with all occurrences of the variable  $x$  (not within  $\text{next}()$ ) replaced by the expression  $e$ , and  $e_1[e/\text{next}(x)]$  is  $e_1$  with all occurrences of  $\text{next}(x)$  replaced by  $e$ . Note that we can never get nested  $\text{next}()$ s by performing these substitutions (i.e. the set of expressions is closed under them).

**Remark 2.5.5 (Substitutivity)**  $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$  does not imply  $\llbracket e_1[e/x] \rrbracket = \llbracket e_2[e/x] \rrbracket$ .

**Example 2.5.6** Here is an example of the failure of substitutivity:

$$\begin{aligned}
 e_1 &= 2 * x \\
 e_2 &= x + x \\
 e &= \{2, 3\} \\
 \llbracket e_1[e/x] \rrbracket &= \llbracket 2 * \{2, 3\} \rrbracket \\
 &= \lambda ss'. \{4, 6\} \\
 \llbracket e_2[e/x] \rrbracket &= \llbracket \{2, 3\} + \{2, 3\} \rrbracket \\
 &= \lambda ss'. \{4, 5, 6\}
 \end{aligned}$$

The reason for the failure is clear: after substituting in  $x + x$ , the non-deterministic expression will occur twice and can evaluate differently the two times. In  $2 * x$  it occurs only once.

Substitutivity holds if  $e$  is deterministic, or  $e_1, e_2$  have just one occurrence of  $x$  or  $\text{next}(x)$ . To prove this, we need the following lemmas. Write  $s_x^v$  for the state just like  $s$  except that  $x$  has the value  $v$ .

**Lemma 2.5.7** Let  $e, f$  be SMV expressions, then

$$\llbracket e[f/x] \rrbracket(s, s') \supseteq \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e \rrbracket(s_x^v, s')$$

Moreover, equality holds if  $f$  is deterministic, or  $e$  has just one occurrence of  $x$ .

**Proof** We prove the lemma by induction on the structure of  $e$ .

- $e = d$ , where  $d$  is a constant.

$$\text{RHS} = \llbracket d \rrbracket(s, s') = \bigcup \llbracket d \rrbracket(s_x^v, s') = \text{LHS},$$

- $e$  is a variable.

If  $e$  is a variable  $y$  different from  $x$ , then the situation is as for constants.

If  $e$  is the variable  $x$ , then

$$\begin{aligned}
 \llbracket e[f/x] \rrbracket(s, s') &= \llbracket f \rrbracket(s, s') = \\
 \{s_x^v(x) \mid v \in \llbracket f \rrbracket(s, s')\} &= \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket x \rrbracket(s_x^v, s')
 \end{aligned}$$

- If  $e = \text{next}(z)$  for any variable  $z$ , again the situation is as for constants.

- $e = e_1 \circ e_2$ .

$$\begin{aligned}
 & \llbracket (e_1 \circ e_2)[f/x] \rrbracket(s, s') \\
 &= \left\{ w_1 \llbracket \circ \rrbracket w_2 \mid w_1 \in \llbracket e_1[f/x] \rrbracket(s, s'), w_2 \in \llbracket e_2[f/x] \rrbracket(s, s') \right\} \\
 &\supseteq \left\{ w_1 \llbracket \circ \rrbracket w_2 \mid w_i \in \llbracket e_i \rrbracket(s_x^{v_i}, s'), v_i \in \llbracket f \rrbracket(s, s'), i = 1, 2 \right\} \\
 &\supseteq \left\{ w_1 \llbracket \circ \rrbracket w_2 \mid w_i \in \llbracket e_i \rrbracket(s_x^v, s'), v \in \llbracket f \rrbracket(s, s'), i = 1, 2 \right\} \\
 &= \left\{ \llbracket e_1 \circ e_2 \rrbracket(s_x^v, s') \mid v \in \llbracket f \rrbracket(s, s') \right\}
 \end{aligned}$$

The first step uses the semantics of  $\circ$ , in the second we apply the induction hypothesis to conclude that  $\llbracket e_i[f/x] \rrbracket(s, s') \supseteq \{w \mid w \in \llbracket e_i \rrbracket(s_x^v, s'), v \in \llbracket f \rrbracket(s, s')\}$ . Then we use rules of sets, and finally the definition of  $\circ$ . We also used the induction hypothesis to conclude that  $\llbracket e[f/x] \rrbracket(s, s') \supseteq \{w_1 \llbracket \circ \rrbracket w_2 \mid w_i \in \llbracket e_i \rrbracket(s_x^{v_i}, s'), v_i \in \llbracket f \rrbracket(s, s'), i = 1, 2\}$ .)

Now suppose  $f$  is deterministic. Then the first two occurrences of  $\supseteq$  above become  $=$  by induction hypothesis, and the third becomes  $=$  by the fact that  $\llbracket f \rrbracket(s)$  is a singleton.

Suppose  $e$  has just one occurrence of  $x$ , say, in  $e_1$ . Then  $e_2$  does not depend on  $x$ . By inductive hypothesis,  $\llbracket e_1[f/x] \rrbracket(s, s') = \bigcup \llbracket e_1 \rrbracket(s_x^v, s')$  (first inclusion). Also  $\llbracket e_2[f/x] \rrbracket(s, s') = \bigcup \llbracket e_2 \rrbracket(s_x^v, s') = \llbracket e_2 \rrbracket(s, s')$ , i.e.  $\llbracket e_2 \rrbracket(s_x^v, s')$  is independent of the choice of  $v \in \llbracket f \rrbracket(s, s')$ , so the middle line becomes  $\{w_1 \llbracket \circ \rrbracket w_2 \mid w_1 \in \llbracket e_1 \rrbracket(s_x^{v_1}, s'), w_2 \in \llbracket e_2 \rrbracket(s, s'), v_1 \in \llbracket f \rrbracket(s, s')\}$ . This justifies  $=$  for the second and third  $\supseteq$ .

- Case statement: similar.

- $e = e_1 \text{ union } e_2$ .

By inductive hypothesis, we have

$$\begin{aligned}
 & \llbracket e_1[f/x] \text{ union } e_2[f/x] \rrbracket(s, s') = \\
 &= \llbracket e_1[f/x] \rrbracket(s, s') \cup \llbracket e_2[f/x] \rrbracket(s, s') \\
 &= \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e_1 \rrbracket(s_x^v, s') \cup \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e_2 \rrbracket(s_x^v, s') \\
 &= \bigcup_{v \in \llbracket f \rrbracket(s, s')} (\llbracket e_1 \rrbracket(s_x^v, s') \cup \llbracket e_2 \rrbracket(s_x^v, s')) \\
 &= \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e_1 \text{ union } e_2 \rrbracket(s_x^v, s').
 \end{aligned}$$

□

## 2.5. The semantics of the SMV feature construct

---

We can prove a similar lemma for substitutions on  $\text{next}(x)$ .

**Lemma 2.5.8** Let  $e, f$  be SMV expressions, then

$$\llbracket e[f/\text{next}(x)] \rrbracket(s, s') \supseteq \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e \rrbracket(s, s'_x{}^v)$$

Moreover, equality holds if  $f$  is deterministic, or  $e$  has just one occurrence of  $\text{next}(x)$ .

**Proof** Again we prove the lemma by induction on the structure of  $e$ .

- $e = d$ , where  $d$  is a constant: as before
- $e$  is a variable: as for constants.
- $e = \text{next}(z)$  for a variable  $z$ .

If  $e = \text{next}(y)$ , for  $y$  different from  $x$ , then the situation is as for constants.

If  $e = \text{next}(x)$ , then

$$\begin{aligned} \llbracket e[f/\text{next}(x)] \rrbracket(s, s') &= \left\{ v \mid v \in \llbracket f \rrbracket(s, s') \right\} = \\ &= \left\{ s'_x{}^v(x) \mid v \in \llbracket f \rrbracket(s, s') \right\} = \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket \text{next}(x) \rrbracket(s, s'_x{}^v). \end{aligned}$$

- $e = e_1 \circ e_2$ .

By a similar argument to the previous proof we find

$$\begin{aligned} &\llbracket (e_1 \circ e_2)[f/\text{next}(x)] \rrbracket(s, s') \\ &= \left\{ w_1 \llbracket \circ \rrbracket w_2 \mid w_1 \in \llbracket e_1[f/\text{next}(x)] \rrbracket(s, s'), w_2 \in \llbracket e_2[f/\text{next}(x)] \rrbracket(s, s') \right\} \\ &\supseteq \left\{ w_1 \llbracket \circ \rrbracket w_2 \mid w_i \in \llbracket e_i \rrbracket(s, s'_x{}^{v_i}), v_i \in \llbracket f \rrbracket(s, s'), i = 1, 2 \right\} \\ &\supseteq \left\{ w_1 \llbracket \circ \rrbracket w_2 \mid w_i \in \llbracket e_i \rrbracket(s, s'_x{}^v), v \in \llbracket f \rrbracket(s, s'), i = 1, 2 \right\} \\ &= \left\{ \llbracket e_1 \circ e_2 \rrbracket(s, s'_x{}^v) \mid v \in \llbracket f \rrbracket(s, s') \right\} \end{aligned}$$

Again the second step relies on the induction hypothesis to conclude that  $\llbracket e_i[f/\text{next}(x)] \rrbracket(s, s') \supseteq \{w \mid w \in \llbracket e_i \rrbracket(s, s'_x{}^v), v \in \llbracket f \rrbracket(s, s')\}$ .

The arguments for the case that  $f$  is deterministic, or that  $e$  has at most one occurrence of  $\text{next}(x)$  are analogous to those in the previous proof.

- Case statement: similar.

- $e = e_1$  union  $e_2$ .

Similar to proof of Lemma 2.5.7.

□

**Corollary 2.5.9** Let  $e, f$  be SMV expressions. If  $f$  does not contain  $\text{next}(x)$ , then

$$\llbracket (e[f/x])[\text{next}(f)/\text{next}(x)] \rrbracket \supseteq \bigcup_{\substack{v \in \llbracket f \rrbracket(s, s') \\ v' \in \llbracket f \rrbracket(s, s')}} \llbracket e \rrbracket(s_x^v, s_x^{v'})$$

As in the lemma, if  $f$  is deterministic, or  $e$  has just one occurrence of  $\text{next}(x)$ , equality holds.

**Example 2.5.10** We give an example of proper inclusion for Lemma 2.5.7. Again let  $e = x + x$  and  $f = \{2, 3\}$ . Note  $f$  is non-deterministic.

$$\text{LHS} = \llbracket (x + x)[\{2, 3\}/x] \rrbracket(s, s') = \llbracket \{2, 3\} + \{2, 3\} \rrbracket(s, s') = \{4, 5, 6\}.$$

$$\text{RHS} = \bigcup_{v \in \{2, 3\}} \llbracket x + x \rrbracket(s_x^v, s') = \{4, 6\}.$$

**Corollary 2.5.11** If  $e$  is deterministic, or  $e_1, e_2$  have just one occurrence of  $x$ , then

$$\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \text{ implies } \llbracket e_1[e/x] \rrbracket = \llbracket e_2[e/x] \rrbracket.$$

The previous lemmas and examples show that the SMV language of non-deterministic expressions, although simple and intuitive, fails a property of substitutivity. This property is important to us because the *treat* feature is defined in terms of substitution, and we would like to have this property in order to guarantee that the *treat* feature is nicely behaved.

Our first approach to this problem was to restrict to the cases of lemmas 2.5.7 and 2.5.8 in which  $f$  is deterministic. Looking again at the definition of the feature construct in section 2.3, this would mean that  $f$  and  $\varphi$  in the *treat* feature would have to be deterministic.

We can avoid this restriction, however, by defining substitution in a cleverer way. First note that all the non-determinism in  $f$  can be expressed at the outermost level.

**Lemma 2.5.12** Let  $f$  be a (possibly non-deterministic) expression. Then there are deterministic expressions  $f_1, f_2, \dots, f_n$  such that

$$\llbracket f \rrbracket = \llbracket f_1 \text{ union } f_2 \text{ union } \dots \text{ union } f_n \rrbracket.$$

**Proof** Induction on the structure of  $f$ . (Rewriting the expression is purely mechanical.) □

## 2.5. The semantics of the SMV feature construct

---

The new operator uses this way of rewriting expressions.

**Definition 2.5.13** Let  $e, f$  be expressions and  $x$  a variable. The expression  $e\{f/x\}$  is defined thus:

$$e\{f/x\} = e[f_1/x] \text{ union } e[f_2/x] \text{ union } \dots \text{ union } e[f_n/x]$$

where  $f$  has been written  $f_1 \text{ union } f_2 \text{ union } \dots \text{ union } f_n$  with each  $f_i$  deterministic (see lemma).

Obtaining  $e\{f/x\}$  is easily automated, since it is a straightforward syntactic manipulation to rewrite  $f$  according to Lemma 2.5.12.

Note that we lose some combinations of possible values if a non-deterministic expression  $f$  is substituted for  $x$  in an expression  $e$  with multiple occurrences of  $x$ . For example in  $e = x + x$ , if we replace  $x$  with the expression  $f = 1 \text{ union } 2$  (which non-deterministically yields 1 or 2) as in the above definition, we get  $e\{f/x\} = (1 + 1) \text{ union } (2 + 2) = 2 \text{ union } 4$  which is only a subset of  $f + f = (1 \text{ union } 2) + (1 \text{ union } 2) = 2 \text{ union } 3 \text{ union } 4$ , which would result from straight syntactic substitution ( $e[f/x]$ ).

However, this is typical for substitution in any non-deterministic framework. The previous definition enables us to express the substitutions we need for feature integration.

The following remark using the new substitution operator hints at a criterion for the commutativity of features:

**Remark 2.5.14** Let  $e, f_1, f_2$  be SMV expressions. If  $f_1$  does not contain  $\text{next}(x)$  and  $x$  does not occur in  $f_2$  then

$$(e\{f_1/x\})\{f_2/\text{next}(x)\} = (e\{f_2/\text{next}(x)\})\{f_1/x\}$$

up to reordering of subterms. This also holds for ordinary substitution  $\cdot[\cdot]$ . In the remainder of this chapter, we will usually have  $f_2 = \text{next}(f_1)$ , in which case the remark applies.

Hence substitutions commute if one concerns the current value and the other one the next value of a state variable. (Obviously substitutions on different variables in the current state need not commute.)

With this new operator, we obtain the desired result for substitution:

**Lemma 2.5.15** Let  $e, f$  be SMV expressions. Then

$$\begin{aligned} \llbracket e\{f/x\} \rrbracket(s, s') &= \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e \rrbracket(s_x^v, s') \\ &\text{and} \\ \llbracket e\{f/\text{next}(x)\} \rrbracket(s, s') &= \bigcup_{v \in \llbracket f \rrbracket(s, s')} \llbracket e \rrbracket(s, s_x^v) \end{aligned}$$

**Proof** By induction, using lemmas. □

The other side of substitutivity asks that  $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$  implies  $\llbracket e[e_1/x] \rrbracket = \llbracket e[e_2/x] \rrbracket$ , i.e., substituting equivalent expressions into an expression results in equivalent expressions. (Analogously for  $e[e_1/\text{next}(x)]$ .) This holds without qualification in our semantics:

**Proposition 2.5.16**  $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$  implies  $\llbracket e[e_1/x] \rrbracket = \llbracket e[e_2/x] \rrbracket$  and  $\llbracket e[e_1/\text{next}(x)] \rrbracket = \llbracket e[e_2/\text{next}(x)] \rrbracket$ .

**Proof** Induction on the structure of  $e$ . If  $e$  is a constant or a variable, or  $\text{next}(z)$  for a variable  $z$ , the result is straightforward; otherwise,

- if  $e = f_1 \circ f_2$  then

$$\begin{aligned} \llbracket e[e_1/x] \rrbracket(s, s') &= \llbracket (f_1 \circ f_2)[e_1/x] \rrbracket(s, s') \\ &= \llbracket f_1[e_1/x] \circ f_2[e_1/x] \rrbracket(s, s') \\ &= \llbracket f_1[e_1/x] \rrbracket(s, s') \llbracket \circ \rrbracket \llbracket f_2[e_1/x] \rrbracket(s, s') \\ &= \llbracket f_1[e_2/x] \rrbracket(s, s') \llbracket \circ \rrbracket \llbracket f_2[e_2/x] \rrbracket(s, s') \end{aligned}$$

The last step uses the inductive hypothesis. Now this expression can be packed up again to obtain  $\llbracket e[e_2/x] \rrbracket(s, s')$ .

- Union, case statements, etc: similar. □

Substitutivity will be important for the application to features in a later section.

We return to the main theme of this section, which is defining the semantics of SMV. Having examined the semantics of expressions, we now give the semantics of complete programs. Since expressions do most of the work in SMV programs, there is not much more to do:

**Definition 2.5.17** Assignments denote relations on  $S \times S$ :

- $\llbracket \text{next}(x) := e \rrbracket = \{(s, s') \mid s'(x) \in \llbracket e \rrbracket(s, s')\}$ ;

## 2.5. The semantics of the SMV feature construct

---

- $\llbracket x := e \rrbracket = \{(s, s') \mid s(x) \in \llbracket e \rrbracket(s, s')\}$ .

The transition relation is given by

$$R = \bigcap_{a \text{ an assignment}} \llbracket a \rrbracket.$$

An SMV program  $P$  denotes a pair  $\llbracket P \rrbracket = (S, R)$ , where  $S$  is the set of states (given in definition 2.5.1), and  $R$  is the transition relation. We may now apply this semantics to verify the examples given in section 2.5.2.

### 2.5.3 Semantics of the feature construct for SMV

**Definition 2.5.18 (Admissible SMV programs)** An SMV program is *admissible* if:

- there are no assignments to current variables;
- and in any assignment of the form  $\text{next}(\mathbf{x}) := e$ ,  $\text{next}(\mathbf{x})$  does not occur in  $e$ .

We assume that the base system is an admissible SMV program, and we also make the assumption that, in the features

```

if  $\varphi$  then impose  $\text{next}(\mathbf{x}) := f$ 
if  $\varphi$  then treat  $\mathbf{x} = f$ 

```

the condition  $\varphi$  and the expression  $f$  is deterministic, and that  $\text{next}()$  does not occur in  $\varphi$ .<sup>11</sup> Note that this does *not* mean that we cannot apply features to non-deterministic systems; only that we cannot impose that a variable have a non-deterministic value, or treat it as having one.

As stated before, we do not allow  $\text{next}()$  to occur in treat features: if, in the program, an expression would refer to  $\text{next}(\mathbf{x})$ , then the integration of such a feature would lead to double nexts, i.e. a reference to a successor state of the next state, which cannot be determined from the current state. The semantics given in the previous section determine values in the next state from the current and the next state only. This restriction also means that for treat features we can write  $\llbracket f \rrbracket(s)$  instead of  $\llbracket f \rrbracket(s, s')$ , since  $f$  cannot depend on  $s'$ .

The assumption that the expression  $f$  occurring in the feature is deterministic means that, for any  $s$ ,  $\llbracket f \rrbracket(s)$  is a singleton. We will write  $\llbracket f \rrbracket(s)$  to be the *value* of  $f$  in  $s$ , rather than the singleton set containing that value, in

---

<sup>11</sup>This is not a serious restriction, since  $f$  may still be a conditional expression.

order to simplify notation. It means we can write  $s_x^{\llbracket f \rrbracket(s)}$  for the state like  $s$  but with  $x$  having the value of  $f$  in  $s$ , etc.

For ease of description, we use the symbol  $\delta$  for features using the impose keyword, and  $\varepsilon$  for features using the treat keyword.

**Definition 2.5.19 (Semantics of features)** Let  $A$  be an automaton, i.e. a binary relation on a set of states.

- If  $\delta$  is the feature

if  $\varphi$  then impose next( $x$ ) :=  $f$

then

$$\begin{aligned} \llbracket \delta \rrbracket(A) = & \{(s, s') \mid s \not\vdash \varphi, (s, s') \in A\} \\ & \cup \{(s, s_x^{\llbracket f \rrbracket(s, s')}) \mid s \Vdash \varphi, (s, s') \in A\}. \end{aligned}$$

Thus, we retain transitions  $(s, s') \in A$  which do not trigger the feature ( $s \not\vdash \varphi$ ); in the case that the feature is triggered ( $s \Vdash \varphi$ ) we alter the target state to take account of the impose.

- If  $\varepsilon$  is the feature

if  $\varphi$  then treat  $x = f$

then

$$\begin{aligned} \llbracket \varepsilon \rrbracket(A) = & \left\{ (s, s') \mid s \not\vdash \varphi, s' \not\vdash \varphi, (s, s') \in A \right\} \cup \\ & \left\{ (s, s') \mid s \not\vdash \varphi, s' \Vdash \varphi, (s, s_x^{\llbracket f \rrbracket(s')}) \in A \right\} \cup \\ & \left\{ (s, s') \mid s \Vdash \varphi, s' \not\vdash \varphi, (s_x^{\llbracket f \rrbracket(s)}, s') \in A \right\} \cup \\ & \left\{ (s, s') \mid s \Vdash \varphi, s' \Vdash \varphi, (s_x^{\llbracket f \rrbracket(s)}, s_x^{\llbracket f \rrbracket(s')}) \in A \right\} \end{aligned}$$

Again, we retain transitions  $(s, s') \in A$  which do not trigger the feature. Here, if the feature is triggered, we behave as if  $x$  had the value of  $f$  in the current or next state, respectively. That is, we transition from  $s$  to  $s'$  if there was a transition from  $s_x^{\llbracket f \rrbracket(s)}$  to  $s_x^{\llbracket f \rrbracket(s')}$ . (Recall that we ruled out occurrences of next() in  $f$  as well as non-determinism in  $f$ .)

**Remark 2.5.20** As we already observed on page 26, IF *cond* THEN TREAT  $x = f$  is equivalent to

## 2.5. The semantics of the SMV feature construct

---

```

TREAT  $x = \text{case}$ 
       $\text{cond} : f ;$ 
       $1 : x ;$ 
       $\text{esac}$ 

```

Hence we can rewrite all treat features accordingly, incorporating the condition into  $f$ , and the semantics for a treat feature  $\varepsilon$  reduce to

$$\llbracket \varepsilon \rrbracket(A) = \left\{ (s, s') \mid (s_x^{\llbracket f \rrbracket(s)}, s_x^{\llbracket f \rrbracket(s')}) \in A \right\},$$

which will simplify reasoning about treat features.

The following lemma will be useful in the next proof:

**Lemma 2.5.21** If  $\varepsilon$  is of the form  $\text{TREAT } x = f$ , and  $f$  is a deterministic expression, then

$$\llbracket \varepsilon \rrbracket(A \cap B) = \llbracket \varepsilon \rrbracket(A) \cap \llbracket \varepsilon \rrbracket(B).$$

**Proof**

$$\begin{aligned}
(s, s') \in \llbracket \varepsilon \rrbracket(A) \cap \llbracket \varepsilon \rrbracket(B) & \\
& \Leftrightarrow (\exists v \in \llbracket f \rrbracket(s), v' \in \llbracket f \rrbracket(s'). (s_x^v, s_x^{v'}) \in A) \wedge \\
& \quad (\exists v \in \llbracket f \rrbracket(s), v' \in \llbracket f \rrbracket(s'). (s_x^v, s_x^{v'}) \in B) \\
& \Leftrightarrow (\exists v \in \llbracket f \rrbracket(s), v' \in \llbracket f \rrbracket(s'). (s_x^v, s_x^{v'}) \in A \cap B) \\
& \Leftrightarrow (s, s') \in \llbracket \varepsilon \rrbracket(A \cap B)
\end{aligned}$$

The second step only works if  $\llbracket f \rrbracket(s)$  is a singleton, i.e. if  $f$  is deterministic.  $\square$

Our aim in this section is to show that the semantics of features given above coincides with what SFI actually does. As indicated above, we write  $P + \delta$  for the result of integrating  $\delta$  into  $P$ . Thus, we aim to prove:

**Lemma 2.5.22** Let  $\delta, \varepsilon$  be as above, let  $P$  be an admissible SMV program.

1. If  $P$  is deadlock free and  $\text{next}(x)$  does not occur in  $f$ , then  $\llbracket P + \delta \rrbracket = \llbracket \delta \rrbracket(\llbracket P \rrbracket)$ .
2. If  $f$  and  $\varphi$  are deterministic and contain no occurrences of  $x$  or of the  $\text{next}()$  operator, then  $\llbracket P + \varepsilon \rrbracket = \llbracket \varepsilon \rrbracket(\llbracket P \rrbracket)$ .

**Proof** 1. We show  $(s, s') \in \llbracket P + \delta \rrbracket \Leftrightarrow (s, s') \in \llbracket \delta \rrbracket(\llbracket P \rrbracket)$ .

Suppose  $s \not\models \varphi$ . Then

$$\begin{aligned} (s, s') \in \llbracket P + \delta \rrbracket &\Leftrightarrow (s, s') \in \llbracket P \rrbracket && \text{by construction of } P + \delta \\ &\Leftrightarrow (s, s') \in \delta(P) && \text{by def. of } \llbracket \delta \rrbracket(\llbracket P \rrbracket) \end{aligned}$$

Suppose  $s \models \varphi$ , and suppose the assignment to  $\text{next}(\mathbf{x})$  is  $\text{next}(\mathbf{x}) := e$ . Let  $P'$  be  $P$  without this assignment to  $\text{next}(x)$ . Notice that  $s'_x \llbracket f \rrbracket(s, s') = s''_x \llbracket f \rrbracket(s, s'')$ , since  $s'$  and  $s''$  differ only in their value for  $x$  but  $\text{next}(x)$  does not occur in  $f$ .

$$\begin{aligned} (s, s') \in \llbracket P + \delta \rrbracket & \\ \Leftrightarrow (s, s') \in \llbracket P' \rrbracket \wedge s'(x) \in \llbracket f \rrbracket(s, s') &&& \text{by constr. of } P + \delta \\ \Leftrightarrow \exists s'', f_i \in \det(f). s' = s''_x \llbracket f_i \rrbracket(s, s') \wedge (s, s'') \in \llbracket P \rrbracket &&& \text{(see below)} \\ \Leftrightarrow (s, s') \in \llbracket \delta \rrbracket(\llbracket P \rrbracket) &&& \text{def. of } \llbracket \delta \rrbracket \end{aligned}$$

The middle equivalence is justified as follows:

$\Rightarrow$ . Since  $\llbracket P \rrbracket$  is deadlock free, there is a successor state  $s''$  of  $s$ . Let  $s'' = s'_x{}^v$  for some  $v \in \llbracket e \rrbracket(s)$ . Then, for some  $i$ ,  $s''_x \llbracket f_i \rrbracket(s, s') = s'$  since  $s'(x) \in \llbracket f \rrbracket(s, s')$ . We have  $(s, s'') \in \llbracket P \rrbracket$ . To show  $(s, s'') \in \llbracket P \rrbracket$  it is sufficient to show that it satisfies  $\text{next}(x) = e$ .

$\Leftarrow$ . Suppose  $s''$  is as given. We easily obtain that  $(s, s') \in \llbracket P' \rrbracket$  and  $s'(x) \in \llbracket f \rrbracket(s, s')$ .

2. By remark 2.5.20 we can assume that  $\varepsilon$  has the form  $\text{treat } x = f$ . The expression  $f$  is deterministic and contains no  $x$  or  $\text{next}()$ . This implies that  $\varepsilon$  will not introduce a circular dependency into an assignment  $\text{next}(x) := e$ . Only if  $e$  contains  $\text{next}(x)$ , i.e. if the original assignment is circular, will the resulting assignment be circular.<sup>12</sup>

Now think of  $P$  as the set of its assignments. For each  $a \in P$  we will prove  $\llbracket a + \varepsilon \rrbracket = \llbracket \varepsilon \rrbracket(\llbracket a \rrbracket)$ . Then, by definition of  $\llbracket P \rrbracket$  and the lemma

<sup>12</sup>In fact, it is not possible to prevent circular dependencies completely, since the next values of other variables may depend on  $\text{next}(x)$ . For any single assignment, however, this simple criterion is sufficient.

## 2.5. The semantics of the SMV feature construct

---

above,

$$\begin{aligned} \llbracket \varepsilon \rrbracket \llbracket P \rrbracket &= \llbracket \varepsilon \rrbracket \left( \bigcap_{a \in P} \llbracket a \rrbracket \right) = \\ &= \bigcap_{a \in P} \llbracket \varepsilon \rrbracket \llbracket a \rrbracket = \bigcap_{a \in P} \llbracket a + \varepsilon \rrbracket = \llbracket P + \varepsilon \rrbracket. \end{aligned}$$

Let  $a$  be the assignment  $\text{next}(y) := e$ . We prove  $\llbracket a + \varepsilon \rrbracket = \llbracket \varepsilon \rrbracket(\llbracket a \rrbracket)$ .

The fact that  $f$  is deterministic and contains no  $\text{next}()$  operator will be used in several steps.

$$\begin{aligned} \llbracket a + \varepsilon \rrbracket &= \llbracket \text{next}(y) := e [f/x] [\text{next}(f)/\text{next}(x)] \rrbracket \\ &= \left\{ (s, s') \mid s'(y) \in \llbracket e [f/x] [\text{next}(f)/\text{next}(x)] \rrbracket (s, s') \right\} \\ &= \left\{ (s, s') \mid s'(y) \in \llbracket e \rrbracket \left( s_x^{\llbracket f \rrbracket(s)}, s_x^{\llbracket f \rrbracket(s')} \right) \right\} \\ &= \left\{ (s, s') \mid (s_x^{\llbracket f \rrbracket(s)}, s_x^{\llbracket f \rrbracket(s')}) \in \llbracket a \rrbracket \right\} \\ &= \llbracket \varepsilon \rrbracket(\llbracket a \rrbracket) \end{aligned}$$

First we apply the definition of  $+ \varepsilon$ . We then use the semantics of assignments, apply lemma 2.5.9 and again the semantics of assignments. Finally by remark 2.5.20, we see that  $\llbracket a + \varepsilon \rrbracket = \llbracket \varepsilon \rrbracket(\llbracket a \rrbracket)$ .  $\square$

**Theorem 2.5.23** The feature constructs are syntax-invariant. Let  $P_1, P_2$  be programs and  $\eta$  a feature (could be an **IMPOSE** or **TREAT** feature). Then

$$\llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket \text{ implies } \llbracket P_1 + \eta \rrbracket = \llbracket P_2 + \eta \rrbracket.$$

**Proof** Immediate corollary of the lemma.  $\square$

Note, however, that  $\llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket$  is rather strong: it says  $P_1$  and  $P_2$  denote the same transition system, even on the non-reachable part. It is not possible to relax this condition, since the introduction of a feature may introduce transitions from reachable states to previously unreachable states; in that case, if the non-reachable parts of  $P_1$  and  $P_2$  were different, the reachable state space of  $P_1 + \eta$  and  $P_2 + \eta$  could be different, i.e.  $\llbracket P_1 + \eta \rrbracket \neq \llbracket P_2 + \eta \rrbracket$ .

### 2.5.4 Properties of the feature construct for SMV

**Theorem 2.5.24 (Idempotence of feature addition)**

1. The assignment to  $\text{next}(x)$  in  $P$  is  $\text{next}(x) := e$ , and both  $e$  and  $f$  and  $\varphi$  in the feature  $\delta$  do not contain  $\text{next}(x)$ , then  $\llbracket P + \delta + \delta \rrbracket = \llbracket P + \delta \rrbracket$ .
2. If  $x$  does not occur in the expressions  $\varphi$  and  $f$ , and  $\varphi$  and  $f$  do not contain  $\text{next}()$ , then  $\llbracket P + \varepsilon + \varepsilon \rrbracket = \llbracket P + \varepsilon \rrbracket$ .

**Proof** We omit the semantic brackets  $\llbracket \cdot \rrbracket$  on  $\delta$ ,  $\varepsilon$  and  $P$  for the sake of brevity.

1. Similar to what we did with treat features, we rewrite  $\delta$  to “impose  $\text{next}(x) := c$ ”, where  $c$  denotes the expression

case  
 $\varphi : f$ ;  
 $1 : e$ ;  
 esac;

We prove that  $\delta(\delta(P)) = \delta(P)$ .

Since  $c$  does not mention  $\text{next}(x)$  we know that  $\llbracket c \rrbracket(s, s') = \llbracket c \rrbracket(s, s'_x)$  for any  $v \in \text{type}(x)$ , thus

$$\begin{aligned}
 (s, s') \in \delta(\delta(P)) &\Leftrightarrow s' = s''_x \llbracket c \rrbracket(s, s''), (s, s'') \in \delta(P) \\
 &\Leftrightarrow s' = s''_x \llbracket c \rrbracket(s, s''), s'' = s'''_x \llbracket c \rrbracket(s, s'''), (s, s''') \in P \\
 &\Leftrightarrow s' = s'''_x \llbracket c \rrbracket(s, s'''), (s, s''') \in P \\
 &\Leftrightarrow (s, s') \in \delta(P)
 \end{aligned}$$

2. Again we assume that  $\varepsilon$  has the form “treat  $x = f$ ”. We prove that  $\varepsilon(\varepsilon(A)) = \varepsilon(A)$  for any  $A$ . We write  $\tilde{s}$  and  $\tilde{s}'$  as a shorthand for  $s_x \llbracket f \rrbracket(s, s')$  and  $s'_x \llbracket f \rrbracket(s, s')$ , respectively.

$$\begin{aligned}
 (s, s') \in \varepsilon(\varepsilon(A)) &\Leftrightarrow (\tilde{s}, \tilde{s}') \in \varepsilon(A) \\
 &\Leftrightarrow \left( \tilde{s}_x \llbracket f \rrbracket(\tilde{s}, \tilde{s}'), (\tilde{s}')_x \llbracket f \rrbracket(\tilde{s}, \tilde{s}') \right) \in A \\
 &\Leftrightarrow \left( s_x \llbracket f \rrbracket(\tilde{s}, \tilde{s}'), s'_x \llbracket f \rrbracket(\tilde{s}, \tilde{s}') \right) \in A \\
 &\Leftrightarrow \left( s_x \llbracket f \rrbracket(s, s'), s'_x \llbracket f \rrbracket(s, s') \right) \in A \\
 &\Leftrightarrow (s, s') \in \varepsilon(A)
 \end{aligned}$$

The first and second equivalences are obtained by rewriting; the third and the fourth exploit the fact that  $x$  and  $\text{next}(x)$  do not occur in  $f$  or in  $\varphi$ .  $\square$

## 2.5. The semantics of the SMV feature construct

---

Finally, let us look at when features commute with each other. In general we do not expect that features should commute. However, when they do, it implies a strong form of non-interaction.

Consider the families of features

$$\begin{aligned}\delta_i &= \text{if } \varphi_i \text{ then impose next}(x_i) := f_i \\ \varepsilon_i &= \text{if } \varphi_i \text{ then treat } x_i = f_i\end{aligned}$$

We explore when  $\delta_1$  commutes with  $\delta_2$ , etc.

As usual we rule out features that may lead to circular assignments, i.e. for impose features,  $f_i$  must not refer to  $\text{next}(x_i)$ , and for treat features,  $f_i$  must not refer to  $x_i$  or use  $\text{next}()$ . Also, for both types of features,  $\varphi_i$  must not contain  $\text{next}()$ .

### Theorem 2.5.25

1.  $P + \delta_1 + \delta_2 = P + \delta_2 + \delta_1$ , if  $x_1, x_2$  are distinct variables and  $\delta_1$  does not use  $\text{next}(x_2)$  and vice versa.
2.  $P + \delta_1 + \varepsilon_2 = P + \varepsilon_2 + \delta_1$  if  $x_2$  does not occur in  $\varphi_1$ ,  $x_1$  and  $x_2$  are distinct variables, and  $x_1$  does not occur in  $f_2$  or  $\varphi_2$ .
3.  $P + \varepsilon_1 + \varepsilon_2 = P + \varepsilon_2 + \varepsilon_1$  if:
  - $x_1, x_2$  are distinct variables, and
  - $x_1$  does not occur in  $\varphi_2$  or  $f_2$ , and
  - $x_2$  does not occur in  $\varphi_1$  or  $f_1$ ;

**Proof** For the proof we again assume the simple form of treat features. Note that

$$(s, t) \in \delta_i(A) \Leftrightarrow \left[ \begin{array}{l} s \not\models \varphi_i, (s, t) \in A \\ s \models \varphi_i, t = t'_{x_i} \llbracket f_i \rrbracket^{(s, t)}, (s, t') \in A \end{array} \right]$$

where we use the notation: in square brackets, comma means and, and vertical juxtaposition means or; and

$$(s, t) \in \varepsilon_i(A) \Leftrightarrow (s_{x_i} \llbracket f_i \rrbracket^{(s)}, t_{x_i} \llbracket f_i \rrbracket^{(t)}) \in A$$

1. Expanding  $\delta_1(\delta_2(A))$ , we see that

$$(s, s') \in \delta_1(\delta_2(A)) \Leftrightarrow \left[ \begin{array}{l} s \models \neg\varphi_1 \wedge \neg\varphi_2, (s, s') \in A \\ s \models \neg\varphi_1 \wedge \varphi_2, s = t_{x_2} \llbracket f_2 \rrbracket^{(s, t)}, (s, t') \in A \\ s \models \varphi_1 \wedge \neg\varphi_2, s = t_{x_1} \llbracket f_1 \rrbracket^{(s, t)}, (s, t') \in A \\ s \models \varphi_1 \wedge \varphi_2, s = (t_{x_2} \llbracket f_2 \rrbracket^{(s, t)})_{x_1} \llbracket f_1 \rrbracket^{(s, t_{x_2} \llbracket f_2 \rrbracket^{(s, t)})}, (s, t') \in A \end{array} \right]$$

If  $x_1$  and  $x_2$  are distinct, and  $\llbracket f_1 \rrbracket(s, t)$  does not depend on  $t(x_2)$  and symmetrically  $\llbracket f_2 \rrbracket(s, t)$  is independent of  $t(x_1)$ , then

$$\begin{aligned} (t_{x_2} \llbracket f_2 \rrbracket(s, t))_{x_1} \llbracket f_1 \rrbracket(s, t_{x_2}^{\llbracket f_2 \rrbracket(s, t)}) &= (t_{x_2} \llbracket f_2 \rrbracket(s, t))_{x_1} \llbracket f_1 \rrbracket(s, t) = \\ &= (t_{x_1} \llbracket f_1 \rrbracket(s, t))_{x_2} \llbracket f_2 \rrbracket(s, t) = (t_{x_1} \llbracket f_1 \rrbracket(s, t))_{x_2} \llbracket f_2 \rrbracket(s, t_{x_1}^{\llbracket f_1 \rrbracket(s, t)}) \end{aligned}$$

2. Expanding  $\delta_1(\varepsilon_2(A))$ , obtain

$$(s, t) \in \delta_1(\varepsilon_2(A)) \Leftrightarrow \left[ \begin{array}{l} s \Vdash \neg \varphi_1, \quad (s_{x_2}^{\llbracket f_2 \rrbracket(s)}, t_{x_2}^{\llbracket f_2 \rrbracket(t)}) \in A \\ s \Vdash \varphi_1, \quad t = t'_{x_1} \llbracket f_1 \rrbracket(s, t'), \quad (s_{x_2}^{\llbracket f_2 \rrbracket(s)}, t'_{x_2} \llbracket f_2 \rrbracket(t')) \in A \end{array} \right]$$

Expanding  $\varepsilon_2(\delta_1(A))$ , we get

$$(s, t) \in \varepsilon_2(\delta_1(A)) \Leftrightarrow \left[ \begin{array}{l} s \Vdash \neg \varphi'_1, \quad (s_{x_2}^{\llbracket f_2 \rrbracket(s)}, t_{x_2}^{\llbracket f_2 \rrbracket(t)}) \in A \\ s \Vdash \varphi'_1, \quad t_{x_2}^{\llbracket f_2 \rrbracket(t)} = t'_{x_1} \llbracket f_1 \rrbracket(s), \quad (s_{x_2}^{\llbracket f_2 \rrbracket(s)}, t') \in A \end{array} \right]$$

where  $\varphi'_1$  stands for  $\varphi_1[f_2/x_2][\text{next}(f_2)/\text{next}(x_2)]$ .

$\varphi_1$  holds for the same states in both cases if  $x_2$  does not occur in  $\varphi_1$ . Now, if  $x_1 \neq x_2$  and  $x_1$  does not occur in  $f_2$ , the last line is equivalent to

$$\varphi_1[f_2/x_2], \quad t = t'_{x_1} \llbracket f_1 \rrbracket(s), \quad (s_{x_2}^{\llbracket f_2 \rrbracket(s)}, t'_{x_2} \llbracket f_2 \rrbracket(t')) \in A.$$

3. Expanding  $\varepsilon_1(\varepsilon_2(A))$  and  $\varepsilon_2(\varepsilon_1(A))$  we see that

$$(s, t) \in \varepsilon_1(\varepsilon_2(A)) \Leftrightarrow ((s_{x_1} \llbracket f_1 \rrbracket(s))_{x_2} \llbracket f_2[f_1/x_1] \rrbracket(s_{x_1}^{\llbracket f_1 \rrbracket(s)}), (t_{x_1} \llbracket f_1 \rrbracket(t))_{x_2} \llbracket f_2[f_1/x_1] \rrbracket(t_{x_1}^{\llbracket f_1 \rrbracket(t)})) \in A$$

and

$$(s, t) \in \varepsilon_2(\varepsilon_1(A)) \Leftrightarrow ((s_{x_2} \llbracket f_2 \rrbracket(s))_{x_1} \llbracket f_1[f_2/x_2] \rrbracket(s_{x_2}^{\llbracket f_2 \rrbracket(s)}), (t_{x_2} \llbracket f_2 \rrbracket(t))_{x_1} \llbracket f_1[f_2/x_2] \rrbracket(t_{x_2}^{\llbracket f_2 \rrbracket(t)})) \in A$$

Here we have used the substitution lemma (Lemma 2.5.9) in the form  $\llbracket f_2 \rrbracket(s_{x_1}^{\llbracket f_1 \rrbracket(s)}) = \llbracket f_2[f_1/x_1] \rrbracket(s)$ .

Comparing  $\varepsilon_1(\varepsilon_2(A))$  with  $\varepsilon_2(\varepsilon_1(A))$ , we see that they are equal provided the syntactic substitutions have no effect, i.e. there are no occurrences of  $x_1$  in  $f_2$ , or of  $x_2$  in  $f_1$ . The same condition also ensures that  $f_2$  does not depend on  $x_1$  and  $f_1$  not on  $x_2$ , so that  $\llbracket f_2[f_1/x_1] \rrbracket(s_{x_2}^{\llbracket f_2 \rrbracket(s)}) = \llbracket f_2[f_1/x_1] \rrbracket(s) = \llbracket f_2 \rrbracket(s)$ , and symmetrically.

□

## 2.6 Conclusions

In this chapter, I introduced the feature construct which Mark Ryan and I developed for the SMV model checking system and I demonstrated its use in two sizeable case studies we conducted. The case studies I showed how a wide range of features can be specified using the feature construct. Furthermore, we explored the semantics of the feature construct. The semantics allow some judgements on whether features are going to interact in some way.

The limitations that we encountered in this work were primarily ones of the expressiveness of the SMV language and of model checking, namely the state explosion problem. It became clear that the model checking approach does not scale very well to larger systems. However, even the rather high degree of abstraction in the case studies allowed us to detect a large number of feature interactions which are also found in the real world.

As far as the expressiveness is concerned, we will see in the following two chapters that feature constructs are not restricted to relatively low-level languages like SMV, when I demonstrate feature constructs for PROMELA and CSP. But even for SMV the feature construct proved quite powerful when one takes into account the limited expressiveness of SMV itself. One example for the limitations of SMV and the feature construct was the difficulty of implementing Three-Way-Calling. This feature requires a second copy of the automaton for a telephone to run in parallel with the original transition system, i.e. the product of two telephones, plus a lot of altered transitions to coordinate the two automata. The SMV framework is not flexible enough to achieve this without virtually rewriting the complete phone module in the feature. In CSP or PROMELA this can be achieved much easier.

In the latter part of this chapter, we have seen that the theoretical approach can obviate the actual model checking in several typical cases. Hence we may not need to model check all feature combinations. While the proofs of the theorems may have been cumbersome, theorem 2.5.25 can save significant time and effort, since the preconditions require only syntactic checks on the features. It is likely that in time, more results of this kind will be proven, allowing specifiers and developers to focus their efforts on the analysis of a smaller number of feature combinations. Two types of theorems seem likely:

- theorems about non-interaction of features, based on the variables they use and change (like Theorem 2.5.25), and
- theorems about preservation of properties, depending on the type of property, the variables used in the formula, and the variables changed by the feature.

# Chapter 3

## Features for Promela/Spin

### 3.1 Motivation

The last chapter introduced a feature construct for SMV a simple language for describing synchronous systems. Leading on from the experience gathered with the SMV feature construct, I explore the idea of a feature construct for a richer language, more precisely, one which offers asynchronous processes and primitives for buffered communication.

The language under investigation in this chapter is PROMELA (PROcess Modelling Language) which is used by the SPIN<sup>1</sup> model checker [39], developed by G. Holzmann. PROMELA is an imperative language for specifying asynchronous processes. It offers primitives for both synchronous (i.e. handshake) and buffered communication. Moreover, PROMELA allows the use of local variables as well as global (shared) variables. The model checker SPIN can check a system specified in PROMELA for invariant violations, deadlocks, livelocks, and against arbitrary LTL formulas.

In [16] Cassez proposes a feature construct for PROMELA but his features can merely manipulate the communication between processes. With feature construct I devised, it is also possible to change the computations within a process. Features for PROMELA models resemble *superimpositions*, and, as I discussed on page 15, my feature construct can be seen as a concretisation of the superimposition construct which Katz proposes in [46].

The structure of this chapter is as follows. I first describe the SPIN model checker and its input language PROMELA followed by a short general description of the concept of a feature for a PROMELA program. After that I give a detailed account of my feature construct for PROMELA and of how

---

<sup>1</sup>SPIN is available from <http://netlib.bell-labs.com/netlib/spin/whatispin.html>

features are integrated into a program. This is followed by some examples and a summary of my experiences.

## 3.2 The model checker Spin

Superficially, the model checker SPIN works in a similar fashion to SMV: it takes a program, explores its state space and returns a verdict of true or false for properties that were specified in the program. The mechanics of SPIN differ in many ways from those of SMV; however, the only aspect which is important for us is that temporal properties (apart from deadlock) are specified as PROMELA processes. Alternatively, SPIN can translate an LTL formula to such a process. This test process, or “never claim” in PROMELA terminology, is executed in lockstep with the model under investigation, and the model checker tests whether certain conditions arise in the “never claim” process. Details of the model checking algorithm can be found in [39] and [40].

In addition to general LTL formulas, SPIN can also check assertions on states in a model and search for deadlocks. For any of these three types of properties, SPIN generates a trace on failure of a property, demonstrating how the property can be violated.

## 3.3 The language Promela

A PROMELA program consists of one or more processes, which can communicate through buffered or synchronous channels. Each process is an instance of a process definition, each of which may be instantiated any number of times.<sup>2</sup>

PROMELA’s syntax owes much to the C programming language, with a few notable exceptions. The central elements of a PROMELA program are processes. These are defined through `proctype` declarations. The special process `init` takes the role of `main` in C: this process is executed when the system is run in the simulator or model checker. Other processes are instantiated from `init` (or any running process) when a “`run process`” command is executed.

A `proctype` declaration contains declarations of local variables and a definition of the process’s behaviour in an imperative language. There are two

---

<sup>2</sup>In principle, dynamic instantiation of processes is possible, but the number of processes must be finite to obtain a finite state system. Moreover, SPIN imposes a limit on the number of processes instantiated overall by not recycling process ids and communication channels.

branching constructs, `if ...fi` and `do ...od`. These two are not like any element of the C language. The `if/fi` construct acts as a nondeterministic choice between the branches, each of which is indicated by a double colon (`::`). If a branch begins with a boolean guard, it can be chosen iff the guard is true; if it begins with a send or receive operation, it is enabled iff the operation is possible. (Naturally, this may depend on whether another process is able to engage in such communication.) The `do/od` construct works exactly like a `if/fi` enclosed in an infinite loop, i.e. any time a branch completes its execution, the choice is made anew. The loop is exited only when either a `break` statement is encountered or a `goto` is executed. The `break` and `goto` constructs work just like in C, also in that labels can be placed (almost) anywhere in the code as targets for `goto` commands.

Apart from process definitions, a PROMELA program contains declarations of variables (integer types of various sizes are available) and channels. Each declaration of a channel constant<sup>3</sup> gives a size and a type for the channel, e.g. “`chan out = [1] of int;`” would declare `out` to be a one place buffer accepting `int` values. For handshake communication, one declares a zero sized channel. Channel *variables* can be defined by not ‘assigning’ to the channel name; such variables may be assigned any other channel. Note that it is allowed to pass channels (or rather their ID numbers) along channels. Additionally, there is one special type, `mtype` (“message type”), a user definable enumeration type, which is intended for use in communications for better readability.

In common with many other languages, PROMELA uses “!” to denote send operations (e.g. `out!x`) and “?” for receive operations. PROMELA also offers various tests on channels: `empty`, `full`, tests for the presence of a particular value or message; note that these tests cannot be used on handshake channels. The special wildcard ‘variable’ `_` is used in communications to match any value (“don’t care”).

The syntax of arithmetic and boolean expressions is like in the programming language C, and just as in C arithmetic expression are interpreted as boolean false iff they evaluate to zero. The only difference to the C syntax is PROMELA’s conditional operator “`->`” which replaces “?” to avoid confusion with the receive operation. It is customary to enclose boolean guards in parentheses.

PROMELA statements are separated by semicolons (like in C) or arrows (“`->`”, like in CSP); semantically both symbols are the same. Bracketing of blocks of code is done with `{` and `}`.

There are some more elements in PROMELA that I have not mentioned,

---

<sup>3</sup>In fact, PROMELA does not enforce that these are not assigned to.

like type definitions, arrays and preprocessor directives, and which I will not go into. These are restricted versions their C counterparts.

## 3.4 The feature construct for Promela

A feature usually changes one or more process definitions; it can introduce new global and local variables, and it can add new statements to a definition or change existing ones. It is also possible to define completely new processes in a feature, or to create further instantiations of existing process definitions. Probably the most important aspect is that a feature can change the communication structure, i.e. it may divert some or all messages on a communication channel.

In this section I describe in some detail the syntax of the feature construct and what the different clauses mean when a feature is integrated into a system.

For the purpose of feature integration we make the additional assumption that every process has at most one infinite loop, to which we refer as the *main loop* from now on. Thus every process either terminates after one pass through its sequence of statements, or it enters the main loop after some initialisation code and only leaves it, if at all, to terminate execution. The rationale behind this assumption is discussed in section 3.4.2.

### 3.4.1 Syntax of the feature construct

A feature consists of a `feature` declaration, one or more `roletype` declarations and optionally new `proctype` definitions.

#### The ‘feature’ declaration.

The `feature` declaration (Figure 3.1) names the feature, declares new global variables, and it states which `roletypes` apply to which `proctype` definitions. The `feature` section may also require certain global variables to be present in the system and it can instantiate processes.

The `apply` clause is the centrepiece of the feature: it tells the feature integrator which `proctype` declarations should be modified according to which role-type. The actual changes are then given in the role-type definition.

Currently variables are bound simply by name. The original variables from the `proctype` definition that a feature uses, must be explicitly named in the feature, the specifier must take care to choose new names for new

```
feature name
{
  [ uses global variables ]

  [ new global variables ]

  { apply roletype-name(parameters) to proctype-name; *
  { run proctype-name (parameters); *
}
```

Figure 3.1: The feature declaration

variables. Functions (preprocessor macros and PROMELA inline procedures) are declared like variables.

It would be possible to implement full parametrisation and variable binding as Katz proposes in [46]. This would then replace the `uses` clause in the `roletypes` (see section 3.4.1). This variable binding mechanism would, however, necessitate the renaming of global variables to avoid name conflicts, while at the same time we want to retain the original names for subsequent feature integrations. Since I am more concerned with a proof of concept, I do not deal with these difficulties.

### The ‘roletype’ declarations.

A role-type declaration (Figure 3.2) defines actual changes to processes. The `uses` clause is like the `REQUIRE` of the SMV feature construct: it states that certain local and global variables must be in scope in the `proctypes` it is applied to. (Again functions are declared like variables.) In a parametrised version, all existing variables that a role-type requires would be listed as parameters; they would then be bound in the `apply` clause in the `feature` declaration.

A role-type can declare new local variables (including their initial values; *cf.* `INTRODUCE` section of the SMV feature construct), and its structure offers various ways to add code to or modify the code of the `proctype` it is applied to.

### 3.4. The feature construct for PROMELA

---

```
roletype name(parameters)
{
  [ uses variables ]
  [ new local variables ]

  [ initial code ]
  [ terminal code ]

  {
    [ before (x) code ]
    [ after (x) code ]
    [ afterall (x1, x2, ... xn) code ]

    [ newoption code ]
    [ at_option ([guard]) code ]
    [ at_state (label) code ]
    [ interrupt (guard) code ]

    [ translate x to expr ]
    [ divert ch1 to ch2 ]
    [ filter (ch!pattern) code ]
  }*
}
```

[ ... ] denotes optional elements,  
{ ... }\* stands for 0 or more repetitions of a structure.

Figure 3.2: A roletype definition

**Components of a ‘roletype’ declaration.**

The various clauses that may occur in a `roletype` declaration are listed with short explanations in Table 3.1. The following section explains their meaning in more detail.

The `uses` and `new` clauses should occur in this order at the beginning of the role-type declaration. Apart from that, the components of a role-type declaration may occur in any order; the integrator will apply the corresponding substitutions in the order given. E.g. the result of integrating

```
after(x1) stmnt1; after(x1) stmnt2;
```

will differ from

```
after(x1) stmnt2; after(x1) stmnt1;
```

in that in the first case *stmnt2* will end up before *stmnt1*, while the second case results in the reverse order.

***code*** stands for a single PROMELA statement with a closing semicolon or for a sequence of statements enclosed in ‘{’ and ‘}’. Only the code in the `initial` and `terminal` clauses may contain `run` statements.

In the case of `at_option`, *code* may also contain the placeholder “\$\$” for the code previously given for that branch of a choice construct; see explanations under `at_option` below.

***expr*** is any expression evaluating to a value (in PROMELA all values are of integer type), i.e. anything that SPIN accepts on the right-hand side of an assignment.

***ch*<sub>1</sub>, *ch*<sub>2</sub>**: for the `divert` clause, *ch*<sub>2</sub> can be given by any expression that yields a channel. However, *ch*<sub>1</sub> may not be the result of an expression as, for example, in the code fragment

```
cc[(x==0 -> 1:0)]!x,
```

when *ch*<sub>1</sub> is given as `cc[0]`. In this case the feature integrator could not (at compile time) determine whether or when to divert messages on this channel.

*Remark:* There is a way around this by extending such an array to contain the new channel *ch*<sub>2</sub>. The statement

```
cc[((x==0 -> 1:0)==0 -> 2:(x==0 -> 1:0))]!x
```

would divert exactly the messages on `cc[0]` (= *ch*<sub>1</sub>) to `cc[2]` (= *ch*<sub>2</sub>). But then *ch*<sub>2</sub> may not be given by a conditional expression, – or the

Table 3.1: The components of a roletype

<code>uses variables</code>	<i>variables</i> (local or global) are required in any <code>proctype</code> that the <code>roletype</code> is to be applied to
<code>new local variables</code>	<i>local variables</i> are introduced to the <code>proctype</code> that the <code>roletype</code> is applied to
<code>initial code</code>	<i>code</i> is added at the very beginning of the <code>proctype</code>
<code>terminal code</code>	<i>code</i> is added at the very end of the <code>proctype</code>
<code>before (x) code</code>	<i>code</i> is added directly before assignments to <i>x</i> .
<code>after (x) code</code>	<i>code</i> is added directly after assignments to <i>x</i> .
<code>afterall (x<sub>1</sub>, ..., x<sub>n</sub>) code</code>	<i>code</i> is added after all variables <i>x</i> <sub>1</sub> , ..., <i>x</i> <sub>n</sub> have been assigned to
<code>newoption code</code>	<i>code</i> is added as a new branch in the main reactive loop
<code>at_option ([guard]) code</code>	<i>code</i> is added to every option guarded by <i>guard</i> in the main loop
<code>at_state (label) code</code>	<i>code</i> is added immediately after <i>label</i> .
<code>interrupt code</code>	<i>code</i> is added to the main reactive loop in an <code>unless</code> construct
<code>translate arg to expr</code>	in all send operations and assignments <i>arg</i> is replaced by the expression <i>expr</i> ;
<code>divert ch<sub>1</sub> to ch<sub>2</sub></code>	all send operations on <i>ch</i> <sub>1</sub> are changed to send operations on <i>ch</i> <sub>2</sub> .
<code>filter (ch!pattern) code</code>	all matching send operations on channel <i>ch</i> are <i>replaced</i> by the given code; parts of the <i>pattern</i> can be used in the given code.

expression yielding  $ch_2$  would have to be evaluated in a separate statement, which breaks the atomicity of the send or receive operation. Another problem that this work-around would entail, is that the feature integration itself would introduce assignments to channel variables (and possibly channel “constants”), which would make it impossible to ensure consistency over several integration steps. (*Cf.* section 3.4.2)

It would be most desirable to be able to write send (and receive) statements of the form

$$(cond \rightarrow ch_1 : ch_2)!msg,$$

but SPIN currently does not allow this. In short, the restrictions on  $ch_1$  and/or  $ch_2$  are due to limitations in PROMELA’s grammar.

**guard** can be any statement. In the `at_option` statement, the given guard statement can only be matched syntactically against those present in the code. It would be preferable to perform the matching at run-time, but *guard* may involve a handshake, the executability of which cannot be determined without actually performing the synchronisation.

**arg** in the `translate` clause, stands for a variable or a symbolic constant (i.e. a value of type `mtype`), but not for a channel. (This is again due to the restriction which SPIN/PROMELA places on the syntax for channels.)

**pattern** in a `filter` clause consists of one or more fields, according to the message format defined for the channel  $ch$ . Each field can either be a constant (matching that *explicit* constant), the name of a variable in scope (matching that variable name), or the wild-card “\_” which matches anything in that field, even complex expressions. Anything matched (except for “\_”) can be referred to in the *code* given with the `filter` clause using “ $\$i$ ” where  $i$  is the index of the field, counting from one; “ $\$0$ ” stands for the whole message. As in PROMELA syntax, the two forms of send operations, “ $ch!x, y, z$ ” and “ $ch!x(y, z)$ ”, are considered equivalent, so matching happens independently of the form chosen.

**label** is a standard PROMELA state label in the original program.

Some of the clauses warrant a more detailed description:

**new variable declarations:** Apart from the obvious introduction of completely new variables this is also used to redeclare existing variables

### 3.4. The feature construct for PROMELA

---

with a larger scope, this is especially useful for extending arrays or declaring new constants of type `mtype`. NB: A `roletype` may redeclare *local* variables only.

**afterall** ( $x_1, \dots, x_n$ ) *code*: This clause adds *code* after the values of all given variables have been computed. It regards choice constructs and certain kinds of loops as “atomic” computations (roughly similar to the notion of `atomic` sequences). This requires some understanding of my classification of loops, which I explain in the next section (p. 88).

**at\_option** (*guard*) *code*: As mentioned above, *guard* is textually compared to the guards in the main loop, and the code given replaces the branch(es) with guard given. The guard however remains, and the new code is added after it. In the *code* one can use the symbol “`$$`” for the original code of the branch (excluding the guard).

In the case where no *guard* is given, *code* is used for every option in the main reactive loop, but “`$$`” of course matches each option in turn, so constructions like

```
    :: old guard -> if
        :: (new guard) -> new code
        :: (!new guard) -> skip
    fi
    original code
```

or

```
    :: old guard -> if
        :: (new guard) -> new code
        :: (!new guard) -> original code
    fi
```

can easily be realised.

**interrupt** *code*: *Code* is added to the main reactive loop in an `unless` construct, i.e at the end of the structure

```
do
  :: original code ...
  :: original code ...
od
```

the integrator adds

```
unless{ code }
```

When the integrator encounters an interrupt clause it automatically defines a label `continue` at the beginning of the main loop, so that it is possible to return from the interrupt code with the statement “`goto continue.`”

`at_state (label) code:` *code* is added immediately after *label*.

### New processes.

A feature may introduce any number of *new proctypes*. These may be instantiated from the `init` process by giving the appropriate `run` statements in the `feature` section. These will be added at the *beginning* of the `init` process. On top of that, role-types may introduce `run` statements in their `initial` or `terminal` clauses.

Of course a feature may also create new instances of `proctypes` that are already the original system. This might, for example, be used to add fault-tolerance to a system, so that a result is only accepted by consensus (*cf.* the Byzantine agreement problem). The instantiation of such processes follows the same rules as for new `proctypes`.

## 3.4.2 Restrictions and practical considerations

We assume that every process has the basic structure<sup>4</sup> detailed in Figure 3.3.

In principle, it is possible to integrate any feature into any system that provides `proctypes` with the right names and arities and that has variables matching those required by the feature and its role-types. Obviously, for every process definition we can write a feature that will render it useless; for example, adding an non-terminating loop to the front of the process. We can however give some guidelines to the sort of programs that features can be added to with few side effects. These guidelines facilitate the static analysis of the control flow and of the structure of communications.

- Programs should not contain `goto` statements; these would make the detection of loops impossible. The only exception to this rule are those `gotos` introduced by previous feature through `interrupt` clauses.
- `proctypes` should only have one reactive loop (see “Dealing with loops” below).

---

<sup>4</sup>In [46] Katz too, remarks that, in general one will have to transform processes to a certain form. He specifically points out the existence of a normal form for CSP processes.

```

proctype (parameters)
{
    [ variable declarations ]
    [ other initialisations ]
    continue:
    [
        do
            :: a choice
            :: another choice
            ::
            :: yet another choice
        od
        [ unless { interrupt code [ goto continue ] } ]
    ]
    [ cleanup code ]
}

```

Figure 3.3: Structure of a process in the base system

- generic channel variables (type `chan`) should only be assigned once; channel variables of declared type (i.e. initialised variables) may never be overwritten. Processes should not communicate channels in any way, including via global variables.

The last restriction forbids ‘mobility’. Features and the feature construct still make sense when we allow mobility, but features that can be defined with the features construct as defined above have no means to keep track of changing channel variables. We can relax the restriction provided we make sure, none of the channels that the features change are accessed indirectly, i.e. via another channel variable.

#### Dealing with loops

For the purpose of automatic feature integration I introduce a (somewhat artificial) distinction between two different kinds of loops: reactive loops and computational loops. (This distinction is problematic since it is rather fuzzy, see the paragraph “Borderline cases” below.)

**Computational loops.** A *computational loop* computes one or more values and then terminates, i.e. control moves on to other code in the same

**proctype.** Computational loops end after a (bounded) finite number of iterations, and they are often enclosed in `atomic` or `d_step` sequences because they represent ‘internal’ computations within a component, which take negligible time with respect to communication and synchronisation between different processes.

Ideally, computational loops would not involve communication or global variables; however, this restriction cannot be upheld in real life.<sup>5</sup>

**Reactive loops.** A reactive loop never ends, i.e. it does not contain a `break` statement or a `goto` to a label outside the loop. Moreover, the loop’s guards will usually involve operations on channels or conditions on shared variables.

**Borderline cases.** A typical case of a ‘reactive computational’ loop (or ‘computational reactive’ loop) is the polling of a list of clients at regular intervals, e.g. to collect information about their status. Assuming a correct implementation, this loop should never be blocked (under normal circumstances) and it will terminate after a finite number of iterations to make its result available for further processing.

Another possibility is a ‘computational’ loop which can be interrupted by an external event (message or global variable). Even if that event does not occur, the loop will still end after a finite number of steps. Such a loop can be used to model the occurrence of an interrupt or an exception. (PROMELA’s `unless` construct provides an elegant way of coding interrupts or exceptions.)

A third case is that of a process which can take different roles: inside a never-ending outer loop there are two or more reactive loops which represent the different roles. The process can switch between roles by means of `gotos` or `breaks`, if certain conditions arise (e.g. exception/interrupt). This would introduce a hierarchical structure into the model. For example, a phone might be modelled by an originating and a terminating call model within the same `proctype`: when the phone is idle, the process waits in the main loop until it either spontaneously chooses to originate a call, or there is an incoming call, in which case it enters the loop representing the terminating call model.

**Treating computational loops.** The idea of a ‘computational loop’ is that it encodes a *computation* rather than a *process*, and the result of this

---

<sup>5</sup>This is only partly due to limitations imposed by SPIN and the semantics of PROMELA. It is also a sign that the distinction between computational and reactive loops is rather artificial.

computation is assigned to some variable(s). Therefore a computational loop should be treated like a set of assignments, namely to those variables that are assigned to in the body of the loop.

This is the significant difference between `after` and `afterall`: whereas the application of an `after` inserts code immediately after each matching statement, the code given in an `afterall` clause is inserted only after the loop's closing `od`.

#### 3.4.3 Conditionals and choices

For the application of `afterall` clauses the choice construct (“`if ...fi`”) is treated similarly to (computational) loops: for `afterall`,

```
if
  :: (x==0) -> y=1  (a)
  :: else -> skip
fi  (b)
```

is treated like `y = (x==0 -> 1:y)` (“`->`” replaces the C conditional “`?`”), so the new code is inserted at point (*b*), while an `after` clause would insert code at point (*a*). (The two code fragments above are indeed semantically equivalent if the choice construct is enclosed in an `atomic` or `d_step` sequence.<sup>6</sup>)

#### 3.4.4 Atomic and deterministic sequences

**Sequences in the feature.** The code fragments given in a role-type may contain `atomic` and `d_step` sequences, which will be inserted into the original code unchanged for `initial`, `terminal` and `afterall` clauses. For `after` and `before` clauses, the sequence will be *extended* to include the matching assignment. For `d_step` we have to take a little more care, see below.

**Sequences in the original code.** If feature code is to be inserted immediately before or after an `atomic` or `d_step` sequence, i.e. the matching statement for a `before` or `after` clause is inside the sequence, the code will be added *inside* the sequence. Again we have to treat `d_step` sequences more carefully.

---

<sup>6</sup>This implies that my feature construct does not always respect semantic equivalence! However, there appears to be no way around this problem when working on the syntactic level, since there are usually many (syntactic) expressions with identical semantics. For SMV, Theorem 2.5.23 gets around this problem by a very narrow definition of semantic equivalence. Similarly, we could apply operational semantics to PROMELA and mimic all computations of SPIN when processing a code fragment: then semantic and syntactic equivalence would be the same.

**Deterministic sequences.** These are subject to two exceptions to what was said in the last two paragraphs:

- Since send and receive operations inside a `d_step` sequence can lead to errors, the code from the `before` or `after` clause will be included in the sequence only if it does not contain operations on channels.
- For `afterall` clauses, `d_step` sequences are treated as a single statement, i.e. the code given will always be added after of the sequence. (Again we view the `d_step` sequence as a single computation step.)

When using `filter` and `divert` clauses, or if send and receive operations are part of a `d_step` sequence in the feature code, it is the programmer's responsibility to ensure the resulting code cannot lead to blocking statements inside `atomic` or `d_step` sequences.

#### 3.4.5 Miscellaneous issues

Since `assert` statements serve the verification of the *unfeatured* system, they are removed when a feature is integrated. Of course, new assertions can be introduced with the new code given in the feature construct.

`Printf` statements, which are not essential to verification, but very helpful in simulations, cause some problems: if a `printf` appears directly after a guard in the main loop, it has to be bundled with the guard in the matching for `at_option`, otherwise the new body for that branch might not work. We demonstrate this in the second example (3.5.2). Sadly, this solution will still not resolve all problems of this sort.

## 3.5 Examples of features for Promela programs

In this section we will see how useful the PROMELA feature construct is in practice. Like with SMV, I demonstrate the use of the feature construct on a lift system and a telephone system. This time, however, I give only a couple of examples rather than a proper case study. The practical difficulties of implementing a robust feature integrator are too big, due to the unnecessary restrictiveness of PROMELA's syntax.

### 3.5.1 The lift system

I have modelled a simple lift system using the “Single Button Collective Control” (SBCC) algorithm, which we already saw in section 2.4.1 in the

previous chapter. To recapitulate, there is only one button on each landing and inside the lift one button for each floor. The lift travels in one direction as long as there are requests ahead, then reverses direction.

For each of the buttons inside the lift and on the landings there is a process which may press the button if it is not already activated. The core of the model is, of course, the process which reads these requests and moves the lift up and down accordingly, opening and closing the lift doors as appropriate.

Figure 3.4 gives the central process of the lift system, implementing the SBCC controller. I have left out the global variable declarations and the processes for “pressing” the buttons, which are straight forward. The arrays `car_button[]` and `lan_button[]` represent the buttons, `r_up` and `r_down` flag if there are requests above or below the current position, respectively, and `dir` indicates the current direction. The other variables should be self-explanatory.

In contrast to Cassez [16] I do not model the lift cabin and controller separately, since my feature construct does not require this infrastructure. One could say my model is written with quick and easy implementation in PROMELA in mind, whereas Cassez’s model tries to emulate the physical reality as closely as possible. Moreover, Cassez’s feature integration would not work for my model since it manipulates only communication, hence it needs the lift cabin and the controller to be separate processes communicating with each other.

#### Features for the lift system

I have implemented a few features for the lift system; here I describe the Overload feature, which prevents the lift from moving when it is too full.

The `feature` section introduces a new flag `overloaded` to indicate if the lift is too full, it then instructs the integrator to use the role-type “`stopper`” to modify the `proctype` “`lift`”. Finally it introduces a new `run` statement to the init process: the `proctype` “`in_out`” will be instantiated when we run the model.

The process “`in_out`” serves to simulate people entering or leaving the lift so that it is below or above its capacity. Obviously this should only happen when the doors are open. This new process also introduces an assertion to test if the lift really cannot move when it is overloaded.

The role-type “`stopper`” achieves this by forcing the doors to remain open whenever `overloaded` is true. (Obviously this relies on the fact that the original system does not allow the lift to move with open doors.) Since the code given is a `d_step` sequence, in the resulting code, any assignment to `doors` will be included in this sequence. Hence no other process will be able

### 3.5. Examples of features for PROMELA programs

---

```
proctype lift ()
{
  short i;
  bool r_up, r_down;

  do

  :: i = position+1; r_up = false; next_up = -1;
     do /* compute calls above current position */
     :: (i>=nfloors) -> break
     :: (i< nfloors && !req(i)) -> i++
     :: (i< nfloors && req(i)) ->
        r_up = true; next_up = (next_up==-1 -> i : next_up); i++
     od;

  i = position-1; r_down = false; next_dn = -1;
  do /* compute calls below current position */
  :: (i< 0) -> break
  :: (i>=0 && !req(i)) -> i--
  :: (i>=0 && req(i)) ->
     r_down = true; next_dn = (next_dn==-1 -> i : next_dn); i--
  od;

  if /* update buttons; open/close doors */
  :: (position == next_call) ->
     doors = OPEN;
     assert(req(position));
     car_button[next_call]=0; lan_button[next_call]=0; next_call = -1
  :: else -> doors = CLOSED
  fi;

  dir = ((dir==DIR_UP && r_up) || (dir==DIR_DN && r_down) -> dir : -dir);
  if
  :: (dir==DIR_UP && r_up) -> next_call = next_up
  :: (dir==DIR_DN && r_down) -> next_call = next_dn
  :: else -> skip /* idle */
  fi;

  if
  :: (doors==CLOSED && next_call!=-1) ->
     position = position + dir; /* move toward next_call */
  :: else -> skip
  fi;

  od
}
```

Figure 3.4: The basic lift system

### 3.5. Examples of features for PROMELA programs

---

```
feature Overload
{
  new bit overloaded;

  apply stopper() to lift();

  run in_out();
}

roletype stopper()
{
  uses bit doors;
  after (doors)
  d_step{
    if
    :: (overloaded) -> doors=OPEN
    :: (else) -> skip
    fi}
}

proctype in_out()
{
  uses byte position;
  new short old_pos;

  do
  :: atomic{(doors==OPEN) ->
    if
    :: overloaded=1
    :: overloaded=0
    fi}
  /* NB: state can change here! */
  assert(overloaded -> position==old_pos : 1)
  od
}
```

Figure 3.5: The “Overload” feature for the lift system

to detect that the feature – not the original controller – determines `doors'` value.

Other features for the lift, such as Car Preference (giving calls from inside the lift precedence over those from landings) and Parking (moving the lift to a certain floor when there are no requests pending) are only slightly more complicated but not very difficult to code.

#### 3.5.2 The telephone system

A telephone system differs considerably from the lift system in that it is essentially distributed and therefore relies on communication and synchronisation. I have taken a very simple model<sup>7</sup> of the “Plain Old Telephone System” (POTS). In this simplistic system of four telephones, all that a user (a phone) can do is establish a call to another phone – or get the “busy-tone” if that fails. Also, only the originator of a call may end the call.

##### Features for the telephone system

The first “feature” one might like to add is that both partners in a call can choose to shut down the call. For lack of space I show just a short excerpt from the code for the POTS model (Figure 3.6) and the parts of the feature `SymmEnd` (Figures 3.7 and 3.10) which achieve this aim.

```
:: (state==TCONNECTED) ->
  printf("Phone %d: tconnected(%d).\n", self, partner);
  atomic{
    hup[partner]?_ -> /* wait for partner to hang up */
    event = on; dev = on; phon[self]?x; partner = null;
    state = IDLE
  }
```

Figure 3.6: Excerpt from the code for POTS

---

<sup>7</sup>My model is based on code written by M. Calder and A. Miller, University of Glasgow, *cf.* [13].

### 3.5. Examples of features for PROMELA programs

---

```
at_option (state==TCONNECTED)
  if
  :: $$ /* put original code for this branch here */
  :: atomic{
    hup[self]!1; /* signal hang-up to partner */
    printf("Phone %d: tclose(%d).\n", self, partner);
    event = on; dev = on;
    phon[self]?x; assert (x == partner);
    partner = null; state = IDLE
  }
fi
```

Figure 3.7: Excerpt from the SymmEnd feature

The base system (POTS) uses the one place channels `phon[]` (one for each phone) as semaphores and to indicate which phone is connected to which. The channels `hup[]` (again, one for each phone) are synchronous and serve to synchronise the two participants in a call when one hangs up (in the base system this is always the originator), forcing the other party to hang up eventually. The local variables `self` and `partner` are indices to these arrays of channels, `x` serves as a dummy variable. Obviously, `state` marks the state of the phone call at important points in a call. Finally, `event` and `dev` are of no further interest here.

In this example we can see that `printf` statements need special attention: if the feature integrator grouped the `printf` with the body of the branch and put it in the place marked “`$$`”, the integration would result code like that in Figure 3.8.

In this case the process could easily get into a deadlock by choosing the branch with the `printf` statement, when in fact the partner process might never issue a rendezvous offer on `hup[partner]`. Therefore the `printf` statement has to be grouped with the guard, which yields the code in Figure 3.9.

Note, that this may in some cases still lead to similar problems; in those cases the resulting code has to be edited by hand. I see this as a minor flaw, since `printf` mainly serves debugging purposes and does not play an important part in the semantics of PROMELA.

```
:: (state==TCONNECTED) ->
  if
  :: printf("Phone %d: tconnected(%d).\n", self, partner);
    atomic{
      hup[partner]?_ -> /* wait for partner to hang up */
      event = on; dev = on; phon[self]?x; partner = null;
      state = IDLE
    }
  :: atomic{
      hup[self]!1; /* signal hang-up to partner */
      printf("Phone %d: tclose(%d).\n", self, partner);
      event = on; dev = on;
      phon[self]?x; assert (x == partner);
      partner = null; state = IDLE
    }
  }
```

Figure 3.8: Faulty code excerpt

```
:: (state==TCONNECTED) ->
printf("Phone %d: tconnected(%d).\n", self, partner);
if
:: atomic{
  hup[partner]?_ -> /* wait for partner to hang up */
  event = on; dev = on; phon[self]?x; partner = null;
  state = IDLE
}
:: atomic{
  hup[self]!1; /* signal hang-up to partner */
  printf("Phone %d: tclose(%d).\n", self, partner);
  event = on; dev = on;
  phon[self]?x; assert (x == partner);
  partner = null; state = IDLE
}
```

Figure 3.9: Correct code excerpt

### 3.5. Examples of features for PROMELA programs

---

```
interrupt
atomic{
  hup[((state==0CONNECTED || state==0CLOSE)
      -> partner:NPHONES)]?_ ->
  /* terminating line has shut down call */
  printf("Phone %d: oclear(%d).\n", self, partner);
  event = on; dev = on;
  if
  :: (phon[self]?[eval(partner)]) -> phon[self]?x
  :: (!phon[self]?[eval(partner)]) -> skip
  fi;
  partner = null; state = IDLE;
  goto continue
}
```

Figure 3.10: The `interrupt` clause from the “SymmEnd” feature

In Figure 3.10 we can see the `interrupt` clause from the same feature: here we prepare the originating phone for the possibility that the partner may issue a ‘hang-up’ signal. We use a conditional handshake to ensure that the `interrupt` can only happen when the originating phone is in one of the states `0CONNECTED` and `0CLOSE`. To make sure that we use a *fresh* channel, we redeclare the array of channels `hup[]` in the feature declaration:

```
uses chan hup[];
new  chan hup[NPHONES+1];
```

where `NPHONES` was the original dimension of the array – and the number of phone processes in the model.

There is one minor complication in this code fragment: the `interrupt` code has to test the processes semaphore (`phon[self]`), since originator may have cleared it already, before the ‘hang-up’ signal occurred. One might find this problem by looking at the original code – or indeed, by using `SPIN` to debug the feature.

The code excerpts I have shown here constitute the core of the feature description, the only things missing are some syntactic paraphernalia and the declarations of the variables used.

### Other features for POTS

Other features for the telephone system proved to be of varying degrees of difficulty; namely the Call Forwarding features were very easily implemented, whereas Call Waiting and Call Forwarding proved to be far more involved.

## 3.6 Conclusions

In comparison to SMV, PROMELA has a more complex syntax, and there are no denotational semantics, and operational semantics are only given implicitly through the model checker SPIN. This makes it virtually impossible to define semantics for the feature construct beyond the informal account given here.

However, the fact that SPIN offers both simulation and model checking made it quite easy to develop and test features. The examples I gave also show that the base system does not have to be written with features in mind for the feature construct to be applicable. Hence developers can reuse existing PROMELA models with little or no modification.

In this chapter I have pointed out some problems which are in part due to particularities of PROMELA and SPIN, in part to the nature of feature integration. Most of these problems seem solvable, but in some cases it seems necessary to change the syntax of PROMELA to remove some arbitrary restrictions. In the same vein, there remains some work to be done to find out if the expressiveness of the “pattern matching” suggested here is adequate. (In the two systems I have investigated I had little use for the `filter` clause, since the examples did not involve a sophisticated protocol.)

It is also desirable to extend the expressiveness of the `apply` statement so that the specifier can apply different role-types to instances of the same process definition. This again involves some sort of pattern matching, which should be easy to understand as well as flexible and expressive.

Finally, on a more theoretical point, it would be very interesting to characterise a (non-trivial) subset of PROMELA for which the feature construct does indeed respect semantic equivalence (*cf.* footnote on page 90). Sadly, it seems doubtful that this is achievable, since it is not even clear how semantic equivalence for PROMELA processes should be defined. Unless a formal semantics for PROMELA is defined, any notion of equivalence would be *ad hoc* or at least not verifiable, since SPIN does not offer refinement or equivalence checks.

# Chapter 4

## Features for CSP

### 4.1 Motivation

In the two preceding chapters we have already seen feature constructs for two very different languages. The SMV feature construct was simple but very expressive with respect to the expressiveness of the SMV language. I also developed semantics for that feature construct and was able to prove some results on the level of semantics. With the PROMELA feature construct, on the other hand, I did not get very far. Problems with the base language made it difficult to write a feature integrator.

The CSP language presents none of the problems that we encountered with PROMELA: the semantics of CSP are clearly defined, e.g. in [67], and the grammar of the language does not impose arbitrary restrictions. With regards to expressiveness, CSP is rather richer than PROMELA. Although CSP does not offer buffered communication as a primitive, and it does not allow the transmission of channel names, these aspects can easily be simulated at a modest cost in terms of time and memory. Moreover, both a simulator, PROBE (Process Behaviour Explorer), and a powerful model checker, FDR2 (Failures-Divergences Refinement), are available.<sup>1</sup>

The appeal of CSP as a third ‘case study’, compared to SMV, is that CSP is an asynchronous language and it is far more expressive. For example, SMV does not have primitives for communication between modules. So, CSP has all the advantages of expressiveness of PROMELA and none of the disadvantages (e.g. lack of formal semantics). However, the full language of CSP (with all the derived operators) is considerably more complex than SMV and similar in complexity to PROMELA.

---

<sup>1</sup>Evaluation copies of these tools can be downloaded from <http://www.formal.demon.co.uk>

CSP is extremely well suited to defining distributed systems like the telephone network, which involves a lot of communication and synchronisation. On the other hand for, say, the lift system of chapter 2 it would not offer a significant advantage over SMV because communication is not an important aspect.

This chapter is divided into two distinct parts. After a short introduction to CSP, I first describe a general feature construct for CSP. The second part is taken up by a case study using a different, specialised feature construct. The latter was suggested by the Feature Interaction Contest [52] which formed the framework of the case study.

## 4.2 The process algebra CSP

CSP stands for “Communicating Sequential Processes”<sup>2</sup>. As the name suggests, CSP allows us to describe systems consisting of processes<sup>3</sup> which operate independently and communicate with each other. The language offers a rich set of operators to compose processes. Contrast this with PROMELA, where processes could not be built up hierarchically. Like PROMELA, CSP is in essence an asynchronous language, but its communication primitives are synchronous – buffers must be defined as CSP processes.

In the following sections, I will review the syntax and semantics of CSP, in order to introduce the feature construct for CSP in section 4.4.

### 4.2.1 The syntax of CSP

There are two versions of the CSP syntax, a mathematical notation and a machine readable version,  $CSP_M$ . I will use the mathematical notation mainly when talking about syntax and semantics of CSP and the feature construct, while the machine readable variant,  $CSP_M$ , is more appropriate when I discuss the case study.

Each process may be made up of component processes itself. The atomic processes of CSP are *Stop* and *Skip*, denoting a deadlocked process and successful termination, respectively. Any CSP process is defined using *Stop* and *Skip*, the prefix operator, and the composition operators. Process definitions

---

<sup>2</sup>The process algebra CSP was first presented by C. A. R. Hoare in a paper in 1978, a refined version was published in the 1985 book [38]. An updated book on the subject [67] (by A. W. Roscoe) has recently been published.

<sup>3</sup>In fact, the component process do not have to be sequential; the word “sequential” in the name is a remnant from the early days of CSP.

## 4.2. The process algebra CSP

---

may be recursive, which gives rise to infinite behaviours. CSP offers various operators, the most elementary of which are:

**Stop** the deadlock process, not action possible

**prefix** perform an action  $e$  then behave as  $P$ :  $e \rightarrow P$

**Skip** process representing successful termination; equivalent to  $\surd \rightarrow Stop$ , where  $\surd$  is the event denoting termination

**sequential composition** perform as  $P$  until  $P$  terminates (i.e. performs  $\surd$ ), then continue as  $Q$ , written  $P; Q$

**parallel composition**  $P$  and  $Q$  execute in parallel, synchronising on events in  $X$ , in symbols:  $P \parallel_X Q$

**internal choice** choose to behave as either  $P$  or as  $Q$ , written  $P \sqcap Q$

**external choice** proceed as either  $P$  or as  $Q$ , depending on what the environment chooses:  $P \square Q$

**hiding** behave as  $P$ , but hide all actions in  $X$ , written  $P \setminus X$

**renaming** perform  $P$ , where the relation  $R$  gives the renaming for events  $P[[R]]$

The first two operators (“ $\rightarrow$ ” and “ $;$ ”) enable us to express sequential behaviour; parallel composition provides synchronisation and communication between processes and thus coordination of actions between processes; the internal choice operator models nondeterminism, while external choice depends on inputs from the environment; internal or invisible actions of processes can be modelled by hiding; and renaming allows us to build and compose generic processes like buffers easily.

Events are elements of an arbitrary set, denoted by  $\Sigma$ . Events may be structured like in PROMELA, so that one can send or receive more than a simple value. For example, the prefix  $c!x?y:A$  denotes the sending of  $x$  and, simultaneously, receiving of a value for  $y$  along the suitably typed channel<sup>4</sup>  $c$ ; furthermore the set of values for  $y$  that is accepted in this communication is restricted to the set  $A$ . This prefix therefore denotes a whole set of events

---

<sup>4</sup>Strictly speaking, the notion of “channel” is introduced only in  $CSP_M$ , mainly for technical reasons. Every event is either a channel name (for a simple event) a communication along a channel (for composite events). In  $CSP_M$  channels have to be declared in order to enable type checking.

$\{c.x.y \mid y \in A\}$ . The set of all communication events associated with a channel  $c$  is denoted by  $\{| c |\}$ . An important difference between CSP and PROMELA should be noted: synchronisation is multi-way by default, when composing two processes so that they synchronise on an event, this event remains available for further processes to synchronise with. In PROMELA, on the other hand, synchronisation happens only via two-way channels, and synchronisation of more than two processes requires either the use of global variables or the exchange of several messages.

In addition to  $\Sigma$ , there are two special events,  $\tau$  and  $\checkmark$  (“tick”); the latter signifies successful termination, the former is the “silent action”, used to model internal (i.e. invisible) state changes of a process. This is used, for example, in the resolution of internal choice.

**Example 4.2.1 (Sequential CSP processes)**

$$\begin{aligned} P &= a \rightarrow b \rightarrow c \rightarrow Stop \\ Q &= c \rightarrow Skip \\ R &= a \rightarrow b \rightarrow R \end{aligned}$$

The first process performs the events  $a$ ,  $b$  and  $c$  and then stops (deadlocks), the second can perform  $c$  and then terminate (successfully) which, in the semantics will be indicated by the special  $\checkmark$  event. The third process,  $R$  can repeat the events  $a$  and  $b$  indefinitely: after performing  $a$  it performs  $b$  and then starts over again.

More complex processes are formed using the choice operators  $\square$  and  $\sqcap$ , parallel composition  $\parallel_x$ , and the hiding operator  $\backslash$ .

**Example 4.2.2 (Composition of processes)**

$$\begin{aligned} P &= a \rightarrow b \rightarrow e \rightarrow Stop \\ Q &= c \rightarrow d \rightarrow e \rightarrow Stop \\ R &= P \square Q \\ S &= P \sqcap Q \\ T &= P \parallel_{\{e\}} Q \\ U &= R \backslash \{a, c\} \end{aligned}$$

The processes  $R$  and  $S$  can execute the same traces (sequences of events), namely  $\langle a, b, e \rangle$  and  $\langle c, d, e \rangle$  (or any prefix thereof). However, in the composition  $R$ , the environment, i.e. any other process composed with  $R$ , can

### 4.3. The semantics of CSP

---

decide whether  $P$  or  $Q$  is executed, by offering event  $a$  or  $c$ , respectively. Process  $S$ , on the other hand, chooses internally whether it wants to perform  $a$  or  $c$ , before offering the chosen event to the environment.

The parallel composition  $T$  allows  $P$  to perform  $a, b$  and  $e$ , and  $Q$  to perform  $c, d$  and  $e$ , but forces the two processes to synchronise on  $e$ , the intersection of the two sets. Hence  $T$  may perform events  $a, b, c$  and  $d$  in any order, provided  $a$  precedes  $b$  and  $c$  precedes  $d$ ; only after these four events have occurred is action  $e$  enabled, and both processes  $P$  and  $Q$  execute it simultaneously.

Finally, process  $U$  behaves like  $R$ , only that the first event (either  $a$  or  $c$ ) is hidden, so that one can only observe  $U$  performing  $\langle b, e \rangle$  or  $\langle d, e \rangle$ . Note that this also means, that it is impossible to influence which trace is chosen.

CSP defines various other common operators in terms of the ones introduced here. For example, the interleaving parallel composition  $P \parallel\parallel Q$ , in which the processes  $P$  and  $Q$  perform their behaviours completely independently, is defined by

$$P \parallel\parallel Q \stackrel{def}{=} P \parallel_{\emptyset} Q$$

For details of these and other CSP operators, the reader is referred to [67].

**Example 4.2.3** I will use the following two processes as a running example in subsequent sections:

$$\begin{aligned} P(k) &= c!k \rightarrow P((k+1) \bmod 3) \\ Q(n) &= c?x \rightarrow r!(nx) \rightarrow Q((n+1) \bmod 3) \end{aligned}$$

Process  $P(0)$  successively sends the values 0, 1 and 2 along channel  $c$ , becoming  $P(1)$  and  $P(2)$  in the process, and then returns to its initial state  $P(0)$ .  $Q(n)$  reads a value  $x$  transmitted on channel  $c$  and then sends out the result of multiplying  $x$  by  $n$  along  $r$ . (We assume channel  $c$  to be declared to carry values 0,1,2, and  $r$  values from 0 to 4.)

Note that we have to compose the two processes  $P(0)$  and  $Q(0)$  as in

$$R = P(0) \parallel_{\{c\}} Q(0)$$

before we can actually witness them communicating with each other.

## 4.3 The semantics of CSP

The denotational semantics of CSP are given in one of three models: traces, failures, or failures and divergences, in order of decreasing coarseness. The

operational semantics defined by Scattergood [68] and Roscoe [67], which we will see later, make it possible to execute (simulate) CSP models, while remaining consistent with the denotational semantics. The operational semantics form the basis of the CSP model checker FDR2 [26]. It is these that we will exploit when presenting the semantics of a feature construct for CSP. First, however, I briefly present the denotational semantics, as they are the standard semantics for CSP.

In the simplest semantics, *trace semantics*, a process is denoted by the set of all finite sequences of events that it can perform. In trace semantics it is impossible to detect deadlocks: a process may wait indefinitely before performing the next event. To reflect this fact, the set of traces of a process includes the empty trace  $\langle \rangle$ , and for any trace all its prefixes. (In other words, for any process  $P$ ,  $\llbracket P \rrbracket$  is prefix closed.)

To allow deadlock detection, the *failures model* records, for every trace, which events a process can refuse to perform. Thus a failure is a pair  $(s, X)$ , where  $s$  is a finite trace of the process and  $X$  is a set of events it can refuse after  $s$ . A process is deadlocked after performing a trace  $s$  if  $(s, X) \in \text{failures}(P)$  and  $X$  is the set of all events. Finally, the *failures/divergences model* is even finer, allowing the detection of infinite sequences of  $\tau$  actions, i.e. livelock situations. This is achieved by recording, in addition to the failures, the traces  $s$  of a process such that an infinite sequence of  $\tau$  actions can occur after some prefix of  $s$ .

### 4.3.1 Trace semantics of CSP

In the previous section we have already used an intuition about the semantics of CSP processes. To put this on a firmer footing, here is the definition of the trace semantics of the language as introduced so far. Silent actions are not recorded in traces.

#### Definition 4.3.1 (Trace semantics)

$$\begin{aligned}
 \text{traces}(\text{Stop}) &= \{\langle \rangle\} \\
 \text{traces}(\text{Skip}) &= \{\langle \rangle, \langle \checkmark \rangle\} \\
 \text{traces}(a \rightarrow P) &= \{\langle \rangle\} \cup \{\langle a \rangle \frown t \mid t \in \text{traces}(P)\} \\
 \text{traces}(P \square Q) &= \text{traces}(P) \cup \text{traces}(Q) \\
 \text{traces}(P \sqcap Q) &= \text{traces}(P) \cup \text{traces}(Q) \\
 \text{traces}(P \parallel_X Q) &= \{t \in (\alpha P \cup \alpha Q)^* \mid t \upharpoonright (\alpha P \cup X) \in \text{traces}(P) \\
 &\quad \wedge t \upharpoonright (\alpha Q \cup X) \in \text{traces}(Q)\}
 \end{aligned}$$

### 4.3. The semantics of CSP

---

$$\begin{aligned} \text{traces}(P \setminus A) &= \{t \upharpoonright \Sigma \setminus A \mid t \in \text{traces}(P)\} \\ \text{traces}(P[[R]]) &= \{t[R] \mid t \in \text{traces}(P)\} \end{aligned}$$

where  $s \hat{\ } t$  is the concatenation of the sequences  $s$  and  $t$ ,  $X^*$  stands for all finite sequence over the set  $X$ ,  $\alpha P$  denotes the alphabet of  $P$ , i.e. all events that  $P$  could participate in. The sequence  $t$ , after all entries not in  $X$  have been removed, is written  $t \upharpoonright X$ . And finally,  $t[R]$  denotes the sequence resulting from substituting (renaming) the events in  $t$  according to  $R$ .

The definition of semantics for recursively defined processes requires some knowledge of fixed point theory and is beyond the scope of this brief introduction. Suffice it to say that a recursively defined process  $P$  is the (fixed point) solution of the equation  $\mu p.P = P[(\mu p.P)/p]$ , where the notation  $Q[R/p]$  means the substitution of  $R$  for all free occurrences of  $p$  in  $Q$ .

**Example 4.3.2** The trace semantics of  $P(0)$  from example 4.2.3 are as follows.

$$\begin{aligned} \text{traces}(P(0)) = \{ \langle \rangle, \langle c.0 \rangle, \langle c.0, c.1 \rangle, \langle c.0, c.1, c.2 \rangle, \\ \langle c.0, c.1, c.2, c.0 \rangle, \langle c.0, c.1, c.2, c.0, c.1 \rangle, \dots \} \end{aligned}$$

The traces of  $P(1)$  and  $P(2)$  are similar but shifted left by one or two events, respectively, i.e. starting with  $c.1$  or  $c.2$ , respectively.

The traces of the processes  $Q(n)$  are a little more complicated because they depend on the input on channel  $c$ . For every time the prefix  $c?x$  is encountered there are three possible values for  $x$  to consider. An event  $c.k$  is then followed by the event  $c.nk$ , and  $n$  is increased by 1 (modulo 3). An example trace for  $Q(1)$  is  $\langle c.1, r.1, c.1, r.2 \rangle$ .

The composition  $R = P(0) \parallel_{\{c\}} Q(0)$ , has the following traces:

$$\begin{aligned} \text{traces}(R) = \{ \langle \rangle, \langle c.0 \rangle, \langle c.0, r.0 \rangle, \\ \langle c.0, r.0, c.1 \rangle, \langle c.0, r.0, c.1, r.1 \rangle, \\ \langle c.0, r.0, c.1, r.1, c.2 \rangle, \langle c.0, r.0, c.1, r.1, c.2, r.4 \rangle \\ \langle c.0, r.0, c.1, r.1, c.2, r.4, c.0 \rangle, \langle c.0, r.0, c.1, r.1, c.2, r.4, c.0, r.0 \rangle, \dots \} \end{aligned}$$

As we can see, a value sent on  $c$  is always followed by its square being sent on  $r$ , for the values 0, 1 and 2; the sequence then repeats. All traces of  $R$  are also traces of  $Q(0)$ , since the composition with  $P(0)$  only puts a constraint on  $Q(0)$  but does not add new events.

### 4.3.2 Operational semantics of CSP

The operational semantics give a translation from CSP processes to labelled transition systems (LTSs), where states correspond to process expressions and transitions are labelled with events.

I give a brief summary of the operational semantics for CSP, details can be found in [67, p.158ff]. For the treatment of parametrised processes, we refer the reader to [68].

In the full CSP language, a prefix  $e$ , as in  $e \rightarrow P$ , denotes a set of possible communications. To deal with this, we assume the existence of functions *comms* and *subs*. To quote Roscoe [67, p.160]:

- *comms*( $e$ ) is the set of communications described by  $e$ . For example,  $d.3$  represents  $\{d.3\}$ , and  $c?x:A?y$  represents

$$\{c.a.b \mid a.b \in \text{type}(c), a \in A\}.$$

- For  $a \in \text{comms}(e)$ , *subs*( $a, e, P$ ) is the result of substituting the appropriate part of  $a$  for each identifier in  $P$  bound by  $e$ . This equals  $P$  if there are no identifiers bound (as when  $e$  is  $d.3$ ). For example,

$$\text{subs}(c.1.2, c?x?y, d!x \rightarrow P(x, y)) = d!1 \rightarrow P(1, 2)$$

Note that, if the prefix  $e$  occurs more than once in  $P$ , *subs*( $a, e, P$ ) will only substitute occurrences of  $e$  and parts thereof up to the next binding occurrence of  $e$ , since this obviously requires a new substitution when the corresponding communication is executed.

Like Roscoe, I shall not deal with local variables beyond what is covered by the *subs* function.<sup>5</sup>

Another useful auxiliary function gives the set of initial events for a process:

$$\text{initials}(P) = \{e \mid e \in \Sigma \wedge \exists P'. P \xrightarrow{e} P'\}.$$

For example,  $\text{initials}(Q(0)) = \{c.0, c.1, c.2\}$ .

**Transition rules for CSP processes.** In the notation of the semantics, we use the process  $\Omega$  which is basically equivalent to *Stop* in that it has no transitions.

---

<sup>5</sup>For a treatment of CSP with local variables and parametrised processes, the reader is referred to [68].

### 4.3. The semantics of CSP

---

Deadlock: there is no rule for *Stop*, since it has no transitions.

Successful termination:

$$\overline{Skip \xrightarrow{\checkmark} \Omega}$$

Prefix:

$$\overline{e \rightarrow P \xrightarrow{a} subs(a, e, P)} (a \in comms(e))$$

Internal choice:

$$\overline{P \sqcap Q \xrightarrow{\tau} P} \quad \overline{P \sqcap Q \xrightarrow{\tau} Q}$$

Recursive processes:

$$\overline{\mu p.P \xrightarrow{\tau} P[\mu p.P/p]}$$

External choice:

$$\frac{P \xrightarrow{\tau} P'}{P \sqcap Q \xrightarrow{\tau} P' \sqcap Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \sqcap Q \xrightarrow{\tau} P \sqcap Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \sqcap Q \xrightarrow{a} P'} (a \neq \tau)$$

$$\frac{Q \xrightarrow{a} Q'}{P \sqcap Q \xrightarrow{a} Q'} (a \neq \tau)$$

Sequential composition:

$$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q} (a \neq \checkmark)$$

$$\frac{\exists P'. P \xrightarrow{\checkmark} P'}{P; Q \xrightarrow{\tau} Q}$$

Hiding of a set of events,  $B \subseteq \Sigma$ :

$$\frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{a} P' \setminus B} (a \notin B \cup \{\checkmark\})$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \setminus B \xrightarrow{\checkmark} \Omega}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{\tau} P' \setminus B} (a \in B)$$

Renaming with a relation  $R \subseteq \Sigma \times \Sigma$ :

$$\frac{P \xrightarrow{\tau} P'}{P[[R]] \xrightarrow{\tau} P'[[R]]}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P[[R]] \xrightarrow{\checkmark} \Omega}$$

$$\frac{P \xrightarrow{a} P'}{P[[R]] \xrightarrow{b} P'[[R]]} (aRb)$$

Parallel composition:

$$\frac{P \xrightarrow{\tau} P'}{P \parallel_X Q \xrightarrow{\tau} P' \parallel_X Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \parallel_X Q \xrightarrow{\tau} P \parallel_X Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \parallel_X Q \xrightarrow{a} P' \parallel_X Q} (a \in \Sigma \setminus X) \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel_X Q \xrightarrow{a} P \parallel_X Q'} (a \in \Sigma \setminus X)$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel_X Q \xrightarrow{a} P' \parallel_X Q'} (a \in X)$$

$$\frac{P \xrightarrow{\surd} P'}{P \parallel_X Q \xrightarrow{\tau} \Omega \parallel_X Q} \qquad \frac{Q \xrightarrow{\surd} Q'}{P \parallel_X Q \xrightarrow{\tau} P \parallel_X \Omega} \qquad \frac{}{\Omega \parallel_X \Omega \xrightarrow{\surd} \Omega}$$

## 4.4 A feature construct for CSP

### 4.4.1 Ingredients of the feature construct

Just like the feature constructs for SMV and PROMELA, the CSP feature construct has three parts: the **require**, **introduce** and **change** section.

**The require section** states which processes, events and constants the feature relies on. If the base system does not provide these, the feature cannot be integrated.

```

require  $process_1, process_2, \dots$ 
          $event_1, event_2, \dots$ 
          $type_1, type_2, \dots$ 
          $const_1, const_2, \dots$ 

```

where the  $process_i$  are process names<sup>6</sup>,  $event_i$  are CSP channel declarations, and  $type_i$  and  $const_i$  are type and constant declarations.<sup>7</sup>

<sup>6</sup>In principle such a name could be any process valued expression in CSP.

<sup>7</sup>The types could be omitted at the users risk. For types and set constants, these declarations may describe subsets of the ones present in the base file.

**The introduce section** is used to declare new processes etc. which the feature needs. All declarations in this section are standard CSP declarations and definitions, and they are simply added to the base file.

```
introduce processes
         events
         types
         constants
         assertions
```

In principle we could allow the enlargement of datatypes and sets here, too. However, the implementation of such an operation would pose some problems, especially with respect to the definition of sets (e.g. alphabets for synchronisation) in the original system by set comprehension or enumeration. For the theoretical treatment, however, these issues may be ignored.

**The change section** is the crucial part of a feature definition, which allows to change the behaviour of processes (of the base system). In this section we also define the interaction between processes **introduced** by the feature and processes from the base system.

```
change P(x1, ...)
      [ interpret e as f
      | translate e to f
      | override Q
      ]
```

I will give detailed semantics of these clauses, both in terms of the syntactic manipulation of the base system and as operational semantics in the next section. For now I only give the intended meaning three cases for the **change** clause.

- **interpret** masks values received from the environment:

If  $P(x_1, \dots)$  offers  $comms(e)$  among its initial events ( $comms(e) \subseteq initials(P(x_1, \dots))$ ), we subsequently interpret the values received in such an event as given by the expression  $f$ . For this reason,  $e$  must represent a receive operation. The expression  $f$  may depend on all variables in scope, i.e. on the parameters of  $P$  and on the variables occurring in  $e$ .

- **translate** alters values sent to the environment:

Whenever  $P(x_1, \dots)$ 's initial events contain  $comms(e)$ , we perform the action  $f$  instead of  $e$ . In this case,  $e$  must be a send operation. As for `interpret`,  $f$  may depend on all variables in scope.

- `override` replaces the definition of  $P$  with  $Q$ . I will introduce another keyword, `original`, to allow addressing the original definition of  $P$  in the definition of  $Q$ . The new process definition may use the parameters of  $P$ .

It is important to note that  $P(x_1, \dots)$  can be any process term, i.e. “ $c!1 \rightarrow c?x \rightarrow R(x)$ ” is a valid process name for the change section. Later we will see a way to get around some of the awkwardness that this brings, with the `after` modifier.

For a compact presentation of the semantics, we introduce the following notation: a feature

```
change P(x1, ...)
  interpret e as f
```

is written  $\text{int}(P(x_1, \dots), e, f)$ .

Similarly we abbreviate

```
change P(x1, ...)
  translate e to f
```

to  $\text{trans}(P(x_1, \dots), e, f)$  and

```
change P(x1, ...)
  override Q
```

to  $\text{over}(P(x_1, \dots), Q)$ . The keyword/function `original` is abbreviated to `orig`.

## 4.4.2 Semantics of the feature construct

### Choice of semantics

Traditionally, semantics for CSP are given as denotational semantics in terms of the traces, stable failures and failures/divergences models (*cf.* [38, 67]). Thus one might attempt to give the semantics of features as functions from, say, traces to traces. However, in trace semantics a process  $P$  is represented by a set of traces  $\text{traces}(P) \subseteq \Sigma^*$ , which in turn is built from the sets  $\text{traces}(P_i)$  of its component processes  $P_i$ . This raises a problem, since features naturally refer to processes (usually by name), whereas in trace semantics,

from the set of traces of a process one cannot deduce which component process contributed to which event. Therefore it is not sensible to define feature semantics as a transformation on the set of (sets of) traces. This observation extends to the stable failures model as well as to the failures/divergences model.

**Note:** When discussing the semantics of the SMV feature construct, we did not consider multi-module programs and hence did not have to deal with the effect of features on compositions of modules. Also, the state of the transition system was explicitly given by the values of the state variables. For CSP however, communication is the fundamental notion, and it would be unrealistic to treat every base system as consisting of only one “flat” process, bisimilar to the system in question. Thus we need to define the semantics of the feature construct in a way that is transparent to compositions, so that the effects of a featured process on the system it is part of may be assessed. (*Cf.* the example in section 4.4.3.)

In the light of these difficulties, we define semantics for the feature construct with respect to operational semantics for CSP as defined in [67]. The operational semantics give a translation from CSP terms to labelled transition systems (LTSs). Hence a feature will denote a transformation on LTSs, given by new transition rules for featured processes.

### Operational semantics for the CSP feature construct

In giving the semantics, I will keep as close as possible to W. Roscoe’s presentation in [67]. This requires a few introductory remarks.

We shall assume that when we apply the **changes**, the system already contains all definitions **introduced** by the feature, in other words, we give the semantics for features with an empty **introduce** section. (See section 4.4.3 below.)

### The default rule for features

The idea of features is to change certain aspects of a system while leaving most of it unaffected. Thus a featured system should, by default, behave like the original system unless a feature demands otherwise. This is expressed in the following rule:

$$\frac{P \xrightarrow{a} P'}{\delta(P) \xrightarrow{a} \delta(P')} \quad (*) \quad (4.1)$$

(\*) The side condition looks rather complicated,

$$(\delta = \text{int}(Q, e, f) \vee \delta = \text{trans}(Q, e, f) \vee \delta = \text{over}(Q, R)) \wedge (P \text{ is not } Q)$$

but all it says is that  $\delta$  does not specify a change to the process  $P$  whose next transition we are trying to determine. In other words unless the  $\delta$  specifies a change to  $P$ ,  $P$  behaves as it would by its original definition.

The question that the side condition raises is which notion of equality to use to determine whether  $P$  is  $Q$ . Since feature integration is a syntactic operation and the operational semantics for CSP are syntax driven, it makes sense to use syntactic equality rather than any of the equivalences derived from denotational or algebraic semantics. This ensures that the semantics of the feature construct correspond to the changes that feature integration introduces via the CSP source code. However, we should keep in mind that the equality predicate used here is a *parameter* of feature semantics, which we could change. Thus we could choose a semantic equivalence instead, which would, of course, change the semantics of the feature construct.

This makes sense for languages with rich and well-understood semantics, like CSP, where we even have three denotational semantics, enabling us to influence the “granularity” of equivalence. We might find that some features do not display interactions when using one equivalence but do interact in the context of a different equivalence. For SMV the question of the appropriate equivalence was addressed by Theorem 2.5.23 (page 72); for PROMELA the question was precluded by the lack of formal semantics, so the only equivalence readily available was the syntactic one.

### Clauses manipulating communication

Two of the three clauses for the `change` section deal with communication between processes. The first one, `interpret`, makes a process treat a value it has received as being a different value. The second clause `translates` values a process sends to the environment. One could say the feature pretends to the “outside world” that the process in question is dealing with a different value. As I will demonstrate in section 4.4.3, this corresponds to the meaning of `TREAT` and `IMPOSE` in the SMV feature construct.

Both of these clauses rely on pattern matching to identify the event prefixes to be modified. In the simplest case this could be done by direct comparison of the strings; if we want to integrate a sequence of features, however, the matching would have to allow for renaming of variables and arbitrary expressions in prefixes. The special meaning of the type `channel` in CSP (as implemented in `FDR2` and `PROBE`), offers some help, as long as channels are not passed via formal parameters.

One might think that the `interpret` and `translate` features could be modelled by an override feature using the renaming operator; however, renaming does not allow us to distinguish between sending and receiving. (Cf. section 4.4.3) Furthermore, the operational semantics of CSP would mean that the renaming is propagated through all evolutions of a process. For recursively defined processes, this leads to a stack of renaming operations accumulating.

A note on the notation in the following paragraphs on `interpret` and `translate`: there is a slight syntactic mismatch between Roscoe's presentation and the way prefixes are treated in  $CSP_M$  (e.g. by FDR2): in  $CSP_M$  the first element of a prefix *must* be a channel, and channels are treated differently from datatypes. For example, it is not generally possible to extract the channel component from an event, since it is not a regular value. For this reason, channel  $c$  is mentioned explicitly in the examples, but omitted in the transition rules.

**Shadowing received values: interpret**

```
change P(x)
      interpret c.e as c.f
```

or, in mathematical notation  $\delta = \text{int}(P, c.e, c.f)$ , where  $c$  is a channel and  $c.e$  is an event prefix describing a set of communications on this channel. (Cf. note above.)

**Example 4.4.1** We modify the process

$$Q(n) = c?x \rightarrow r!(nx) \rightarrow Q((n + 1) \bmod 3)$$

from example 4.2.3 with the following feature  $\delta$ :

```
change Q(j)
      interpret c?x as c?(2 - x)
```

After integration we get

$$\delta(Q(n)) = c?x \rightarrow r!(n(2 - x)) \rightarrow \delta(Q((n + 1) \bmod 3))$$

The feature  $\delta$  makes  $Q$  'interpret' the value  $x$  differently: all occurrences of  $x$  are replaced by  $(2 - x)$ . If we know that the values sent on  $c$  are a repeating series of 0, 1 and 2, as in the composition with  $P(0)$ , we see that the feature reverses this sequence as far as  $Q$  is concerned.

**Semantics.** The application of an `interpret` feature replaces the value  $a$  received from another process with the value  $f(a)$ . In other words, the featured process “sees” a different value from that which was transmitted from the environment. This leads us to the following rule:

$$\frac{P \xrightarrow{a} \text{subs}(a, e, P')}{\delta(P) \xrightarrow{a} \delta(\text{subs}(f, e, P'))} \quad \begin{matrix} (\delta = \text{int}(P, e, f), \\ a \in \text{comms}(e)) \end{matrix} \quad (4.2)$$

**Sending altered values: translate**

```
change P(x)
      translate c.e = c.f
```

or, in mathematical notation  $\varepsilon = \text{trans}(P, c.e, c.f)$ . Again,  $c.e$  is a prefix denoting a set of communications. However, to be affected by a `translate` feature, it must be a send operation or a handshake. Another way to say this is to demand that  $\text{comms}(c.e)$  is a singleton set.<sup>8</sup> Again, we have made the channel  $c$  explicit for the reasons stated on page 114.

**Example 4.4.2** Let us modify the process

$$P(k) = c!k \rightarrow P((k + 1) \bmod 3)$$

from example 4.2.3 with the feature  $\varepsilon$ :

```
change P(x)
      translate c!x = c!(2 - x)
```

This yields

$$\varepsilon(P(k)) = c!(2 - k) \rightarrow \varepsilon(P((k + 1) \bmod 3))$$

In the composition  $R = P(0) \parallel_{\{c\}} Q(0)$ , this has the same effect on the output from  $Q(n)$  as the `interpret` feature  $\delta$  above would have: the sequence of numbers transmitted on  $c$  is reversed, so  $Q(n)$  reads values in the order 2, 1, 0 (repeating). However, this time the environment sees this reversed sequence sent on  $c$ , whereas with  $\delta(Q(n))$ , the only change visible to the environment was the changed output of  $Q(n)$ . With the `translate` feature  $\varepsilon$  any other process that communicates with  $P$  (on channel  $c$ ) will also be affected by the changes to  $P$ .

---

<sup>8</sup>In the general case, with parameters and local variables, we have to be more careful:  $\text{comms}(c.e)$  must be a singleton set for any (permitted) valuation for the local variables and parameters.

**Semantics.** The effect of a `translate` feature is that the environment of  $\text{trans}(P, a, f)(P)$  sees  $f$  when  $P$  transmits the value  $a$ , as expressed in the following rule. (As before, we do not mention the channel, see explanation on page 114.)

$$\frac{P \xrightarrow{a} \text{subs}(a, e, P')}{\delta(P) \xrightarrow{f} \delta(\text{subs}(a, e, P'))} \quad \left( \begin{array}{l} \delta = \text{trans}(P, e, f), \\ \text{comms}(e) = \{a\} \end{array} \right) \quad (4.3)$$

The intended use is that we only substitute  $a$  with  $f$  in send operations. For this reason we demand that  $\text{comms}(e)$  is the singleton set  $\{a\}$ .<sup>9</sup> The use of  $\text{subs}$  in the above rule serves only to stress the fact that the process resulting after the transition is *not* changed, and to specify (in the side condition) that  $\text{comms}(e)$  is a singleton.

Obviously, care must be taken that  $f$  can actually be received by other processes that were (originally) expecting value  $a$ . A sufficient condition to guard against type and range errors is  $\text{range}(f) \subseteq \text{comms}(e)$  for all prefixes  $e$  of  $\text{type}(a)$  that occur in the system. Nonetheless the featured system may still deadlock if the processes participating in this event do not agree on a value. This, however, is a matter for the designer of the feature.

### Overriding process definitions

Features are often concerned with monitoring certain events to trigger new actions at appropriate points in time. Therefore it makes sense to allow a feature to add components to existing composite processes using any CSP composition operation. For this we define a “catch all” feature clause:

`change`  $P$   
`override`  $Q$

or  $\delta = \text{over}(P, Q)$ .

**Semantics.** The semantics of an `override` feature are given by

$$\frac{Q \xrightarrow{a} Q'}{\delta(P) \xrightarrow{a} \delta(Q')} \quad (\delta = \text{over}(P, Q)) \quad (4.4)$$

Note that this means the feature is not eliminated when  $\delta(P)$  proceeds with the event  $a$  as  $Q$  would but keeping  $\delta(P)$ .

---

<sup>9</sup>Since we do not deal with local variables,  $e$  will be a constant in  $P$  iff  $e$  denotes a send or handshake, as any occurrence of local variables must have been substituted by previous applications of  $\text{subs}$ .

**Accessing definitions of the base system: original**

The keyword `original` allows the user to use the original, unfeatured definition of a process.<sup>10</sup> This is especially useful in overriding features, where we replace a feature definition. It may also be used for formulating assertions (e.g. refinement checks) about a featured and an unfeatured process.

When performing a sequence of integrations, the modifier `original` always refers to the system before the integration of the current feature. The defining property of `original` with respect to a feature  $\delta$  is

$$\frac{P \xrightarrow{a} P'}{\delta(\text{orig}(P)) \xrightarrow{a} \delta(P')} \quad (4.5)$$

Note that this rule is almost the same as the default rule (4.1), and could in fact be integrated with it by defining  $\llbracket \text{orig}(P) \rrbracket = \llbracket P \rrbracket$  but letting  $\text{orig}(P) \neq P$  in the equality test. Textually this is effected by taking the original definition of  $P$  for  $\text{orig}(P)$  but replacing all occurrences of processes (including  $P$ ) therein with their featured versions. Assume, for example, a system where  $P \xrightarrow{a} P' \xrightarrow{b} P$  and  $Q = c \rightarrow \text{Stop}$ . If we integrate  $\delta = \text{over}(P, \text{orig}(P) \square Q)$ , we want the system to offer the choice of  $c$  every time that we are in the state representing  $\delta(P)$ . Hence the execution  $\delta(P) \xrightarrow{a} P' \xrightarrow{b} \delta(P)$  must be possible. This we achieve by substituting in the way described above: in  $P'$ , the occurrences of  $P$  are replaced by  $\delta(P)$ .

Eventually, one may want to implement a kind of version control system to be able to address “original” processes of earlier integration steps.

**Syntactic sugar: after**

In practice it may be desirable to have more ways of specifying the location at which a feature should be applied. I suggest the keyword `after` as a further syntactic element for `change` clauses.

Used as a modifier in `change` clauses, `after e` allows the user to address the parts or sub-processes of  $P$  that follow an event described by prefix  $e$ . During integration, “`change P after e clause`” is expanded to a set of changes where  $P$  is replaced by  $e \rightarrow P_i$  for each  $P_i$  with the property:

$$\exists P'. P \xrightarrow{\Sigma^*} (e \rightarrow P_i)$$

An important detail of the `after` modifier is its interaction with `original`. Since the processes that are actually changed by the feature are the  $P_i$ , in an `after` clause,  $\text{orig}(P)$  refers to  $P_i$ .

<sup>10</sup>This refers only to the last integrated feature, and only to the named process, not to its components.

Since we are thus creating several clauses to integrate, we have to specify how this is to be done in one integration step. The above implies that all occurrences of  $e$  in each possible evolution of  $P$  will be considered. However, these substitutions will concern disjoint sequential parts of the process, since each substitution stops at the next binding occurrence of  $e$ . Hence all the changes that are to be carried out will be independent of each other, and therefore they may be applied in any order. Generally though, the specifier of a feature should beware that the **after** modifier does generate a list of changes.

#### Enlargement of sets and datatypes

This can be achieved by defining a new set constant for the expanded type<sup>11</sup> and the appropriate new channels, and by using the CSP renaming operator to map events involving the original type into such that use the expanded type. However, maintaining clear and consistent naming of sets and types will be very difficult unless the integrator tool can completely control this process. For a commercial tool, one would couch this in further language constructs of the feature construct.

#### 4.4.3 The semantics of feature integration

The full semantics of the feature construct are given by integrating the feature and then applying the extended operational semantics, which define the execution(s) of a process in the presence of features.

The **require** section of a feature merely stipulates some constraints on the systems the feature may be applied to. The **introduce** section adds new processes and events, which in itself will change the semantics of the original system only insofar as new datatypes, constants, events and processes are added. This leads to an extension of the domain into which the system is mapped by  $\llbracket \cdot \rrbracket$ . Before the **changes** are applied, the model of the original system is simply embedded in the extended model.

Only the **change** section affects the execution of the system, and it does so in two ways. In the integration process, the definition of an old process  $P$  is changed, and the new definition is used in all subsequent occurrences of  $P$ ; written as a formula:

$$System' = System[\delta(P)/P],$$

where  $System$  is the original system and  $System'$  is the featured system. Note that for the substitution to be interpreted as a textual operation,  $System$

---

<sup>11</sup>In FDR2 datatypes are treated as sets.

needs to be suitably expanded, so that all occurrences of  $P$  are explicit, including those in recursive definitions.

#### 4.4.4 Miscellaneous issues

**Example 4.4.3 (Global variables)** In CSP a global variable (i.e. reference to a memory location) is modelled as:

```
VAR_empty = write?value -> VAR(value)
VAR(value) = write?new_value -> VAR(new_value)
            [] read -> return!value
```

We can achieve the symmetry that **TREAT** and **IMPOSE** for SMV exhibit, since we have both a mechanism to shadow the value received along **write** (using **interpret**) and to substitute the value sent along **return** (using **translate**). For example the SMV feature

```
CHANGE
MODULE main
    TREAT VAR = e
```

corresponds to the following CSP feature

```
change VAR(value)
    translate return!value = return!e
```

Note that for **interpret** we have to change both processes: **VAR(value)** and **VAR\_empty**.

In fact we have even more control over the interpretation of the variable since we can apply an **interpret** or a **translate** feature either to the process representing the variable, or apply it to some of the processes accessing the variable. In the latter case we can give different “views” of the variable to different processes.

#### Indexed external choice

The manipulation of communication between CSP processes by **translate** and **interpret** features raises one problem which cannot be resolved syntactically:

$$\square_{v \in V} c!v \rightarrow P \quad \text{is equivalent to} \quad c?v:V \rightarrow P$$

in all denotational semantics for CSP. Indeed the latter may be seen as a convenient shorthand for the former. However, only  $c?v:V \rightarrow P$  is recognised as a receive operation as far as feature integration is concerned. The operational semantics of features presented above observe the distinction between the indexed external choice and the receive operation. Therefore feature specifiers must make sure, that they don't fall foul of this distinction of semantically indistinguishable terms in the base system. For the same reasons, one should avoid using indexed external choice to model receive operations in feature definitions.

## 4.5 Feature interaction contest

The second Feature Interaction Contest was held in conjunction with the 6th International Workshop on Feature Interactions in Telecommunication and Software Systems (FIW'00) [53]. The aim of the contest was to compare various methods and tools for detecting feature interactions. To enable a comparison, the contest's objective was to detect interactions among a given set of features for a given telephone system. The contest instructions contained detailed (albeit imprecise) specifications of the base system and twelve features.

In the rest of this chapter I describe how I tackled the contest and the experiences I made in the process. In the following two sections I briefly describe how the contest was set out and what implications that had for detecting feature interactions. I then detail the methods I used and the results I obtained before summing up my experiences in section 4.9.

I used a combination of two techniques: static (syntactic) analysis and model checking. While I had initially planned to integrate the features into the base system and then to detect interactions by model checking, it became clear very quickly that a simple syntactic analysis of the features was sufficient to detect a large number of interactions. Hence I first describe this syntactic method (section 4.6.2), before showing how I used a feature construct and model checking to find more interactions (section 4.7). Finally, I give an overview of the full results of my analysis in section 4.7.4, a complete and detailed description of the results can be found in [65].

## 4.6 The contest model

The base system for the contest is a model of the plain old telephone service (POTS) given as a labelled transition diagram (Figure 4.1) for a single

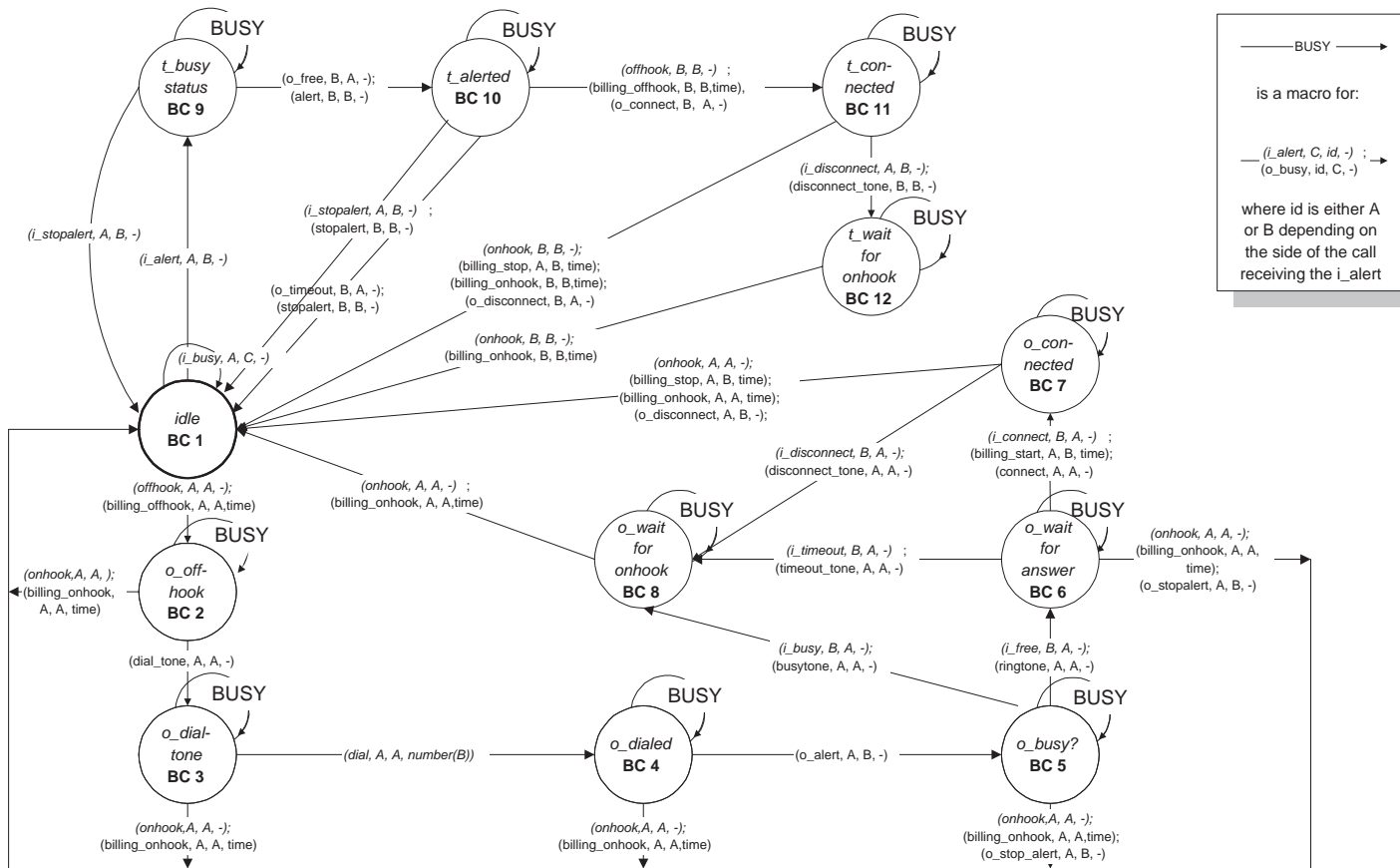


Figure 4.1: The Basic Call Model [52]

#### 4.6. The contest model

telephone line (basic call model, BCM) and a description of the network architecture, a simple star network, with a single switch relaying information between lines, and a billing database, of which no details are given in the contest instructions [52].

Most transitions are labelled with multiple messages. Some of these messages denote local events, such as user input or signals to the user, others stand for messages sent to or received from other phones in the network. Finally, there are billing messages, which are supposed to be logged by some software to obtain billing records. In the diagram, incoming messages are prefixed by “i\_”, outgoing messages by “o\_” and billing messages by “billing\_”. Messages with no prefix represent local events. Messages with no prefix represent local events.

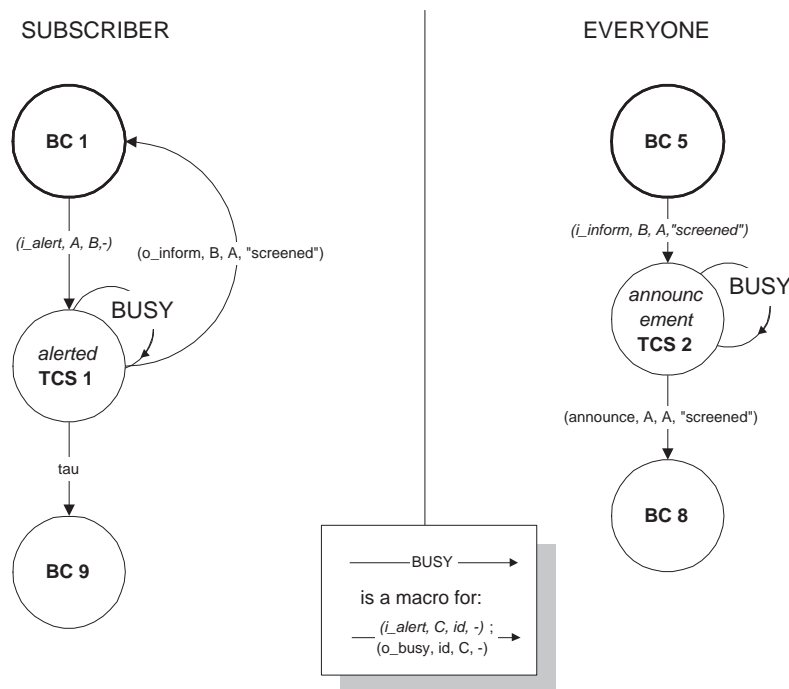


Figure 4.2: Feature Terminating Call Screening [52]

The features, too, are given as labelled transition diagrams, with some BCM states and (usually) several new states, which I will refer to as feature states. Feature integration is defined by adding or replacing, for a given basic call state, the transitions which are given in the feature definition. Here, a transition is replaced if the feature’s transition has the same triggering event as the corresponding transition in the basic call; otherwise the feature’s transition is added. In the diagram for the feature Terminating Call Screening

(Figure 4.2), for example, the transition from the idle state to BC9 of the basic call is replaced by the left diagram (for the subscriber), while in all basic call processes a new transition is added to the state BC5 (“o\_busy?”). This transition is labelled with the message (i\_inform,B,A,“screened”) and leads to the new state TCS2.

The contest comprised the following features:

- CFB** Call Forward on Busy – divert calls when the line is busy
- CNDB** Calling Number Delivery Blocking – do not display the callers number to the callee
- CT** Call Transfer – transfer an active call to another telephone
- CW** Call Waiting – allow subscriber to take a second call and switch between first and second call
- GR** Group Ringing – make three telephones ring for an incoming call to one of them, stop the ringing when one of them is answered
- RBWF** Ring Back When Free – ring back callers who got the busy tone (also known as Automatic Call Back)
- RC** Reverse Charging – callee pays for call
- SB** Split Billing – callee pays part of the call
- TCS** Terminating Call Screening – block calls from certain phones to the subscriber’s phone
- TL** TeenLine – a PIN must be entered before calls can be made
- TWC** Three-Way Calling – allow the subscriber to initiate a second call and let all three parties talk with each other (also known as Conference Call)
- VM** Voice Mail – callers can leave a message if the phone is not answered (also known as CallMinder)

### 4.6.1 Observations

Analysing the feature definitions, I realized that the model given in the contest instructions was extremely prone to interactions. With a little practice I could anticipate many interactions by just looking at the features’ diagrams. To explain how this worked, I classify interactions according to their causes:

#### 4.6. The contest model

---

1. one feature overrides trigger of the other feature (e.g. TCS & GR)
2. one feature bypasses trigger of the other feature (e.g. TCS & CW)
3. one feature sends a message that the other feature cannot process (e.g. TCS & TWC)
4. other causes (e.g. TCS & RBWF, RC & RBWF)

In the first two cases one of the features is not invoked when it should be; in the latter two, both features are active but interfere with each other in some way.

Interactions of type 1 are the easiest to detect: both features are triggered in the same (basic call) state by the same event. Due to the method of feature integration, the feature integrated later overwrites the transition introduced or altered by the earlier one. Interactions are usually serious because they are due to some fundamental incompatibility. In this case the interaction can only be resolved by limiting or refining the behaviour of one or both features.

Type 2 interactions are also not hard to recognise. Whenever a triggering message for one feature can occur in a feature state of another feature, this may lead to the former feature not being invoked even though it should. In the contest, all interactions of this type occurred because of the method of feature integration, and could clearly be avoided by a design that did not put so much emphasis on states. It is very likely that some of the type 2 interactions would then become type 4 interactions, i.e. the type 2 conflict indicated a serious interaction but for trivial reasons.

Interactions of the third type occur when a feature introduces a new message to the system, which may be received in a feature state of another feature. Since only basic call states are altered to be able to handle the new message, a feature on the receiving end will not be able to handle the new message. For these interactions, the same hold as for type 2: in a better, feature-oriented architecture, they would not occur.

The remaining interactions (type 4) are particularly interesting from the point of view of feature interaction detection. They indicate deeper problems in the way that features affect processing and distribution of information in the telephone network. There are no generic tests to detect such interactions, however they violate some 'sensible assumptions' about the working of the system. Furthermore, they only surface in the actual execution of the system, so they are not amenable to static analysis methods, such as proposed in the following section.

### 4.6.2 Static analysis

The observations above prompted me to look for syntactic criteria to detect the first three classes of interactions. I introduce the following notation.

Let  $S$  be the set of all states (both basic call and all possible feature states), and  $E$  the set of all events (messages). To simplify the presentation we omit the subscriber parameters unless absolutely necessary.<sup>12</sup> Feature  $n$  is denoted by  $F_n$ ;  $Sys$  may be a feature, or the Basic Call possibly augmented with a number of features.

$\text{Trans}(Sys) \subseteq S \times E$	one pair $(s, e)$ for each transition in $Sys$ , such that in state $s$ , event $e$ commences the transition
$\text{Triggers}(F_n) \subseteq S \times E$	the basic call states in which the $F_n$ can be triggered, with the corresponding triggering event
$\text{Msgs}(Sys) \subseteq E$	all $i\_xxx$ and $o\_xxx$ messages that appear on transitions of $Sys$ (ignoring the prefixes “ $o\_$ ” and “ $i\_$ ”)

Note that  $\text{Msgs}(Sys)$  contains information about all transitions, while  $\text{Triggers}(F_n)$  only records the new or altered transitions from basic call states introduced by  $F_n$ . Let  $\pi_2(s, e) = e$  denote the projection onto the second component, i.e. events.

**Example 4.6.1** For the Terminating Call Screening feature we have:

$$\begin{aligned} \text{Trans}(\text{TCS}) &= \{(\text{BC1}, i\_alert), (\text{BC5}, i\_inform), \\ &\quad (\text{TCS1}, i\_alert), (\text{TCS1}, o\_inform), (\text{TCS1}, \text{tau}), \\ &\quad (\text{TCS2}, \text{announce}), (\text{TCS2}, i\_alert)\} \\ &\text{and} \\ \text{Triggers}(\text{TCS}) &= \{(\text{BC5}, i\_inform), (\text{BC1}, i\_alert)\} \end{aligned}$$

### Finding interactions

With this notation, the following criteria characterise the first three classes of interactions given on page 123.

1. Later feature overrides earlier one:  
 $\exists (s, e) \in \text{Triggers}(F_1) \cap \text{Triggers}(F_2)$

---

<sup>12</sup>To be fully correct, one would need to take account of several variables, including at least subscriber and partner, for each state.

2. One feature bypasses a trigger of another feature:  
 $\exists e \in \pi_2(\text{Triggers}(F_1)) \cap \pi_2(\text{Trans}(F_2) \setminus \text{Triggers}(F_2))$
3.  $F_1$  may send a message which  $F_2$  cannot handle (“Message not understood”):  
 $\exists e \in \text{Msgs}(F_1) \setminus \text{Msgs}(F_2)$

These criteria will flag some potential interactions that *cannot* occur during normal execution of the featured system. In my experience with the contest model, however, they were quite accurate.

It is important to note that these criteria point to *causes* for interactions. There may be more than one actual, observable interaction for the same event or state-event pair satisfying one of the criteria.

**Example 4.6.2** Since the Group Ringing feature, like TCS, is also triggered by an “i\_alert” message in state BC1, we get a type 1 interaction:

$$\text{Triggers}(\text{TCS}) \cap \text{Triggers}(\text{GR}) = \{(\text{BCi}, \text{i\_alert})\}$$

Thus we can conclude, when TCS and GR are added to the same subscriber’s telephone, whichever feature is added later will disable the earlier one, since it will override the triggers for the feature added earlier.

Furthermore we also get a type 3 interaction, since

$$\text{o\_inform} \in \text{Msgs}(\text{TCS}) \quad \text{but} \quad \text{i\_inform} \notin \text{Msgs}(\text{GR}).$$

Note that a message “o\_xxx” becomes “i\_xxx” for the receiving phone, hence the third criterion says, *if* there is a situation in which the TCS feature sends an “o\_notify” while the other phone is in a GR feature state, then that message cannot be processed (by the GR feature), leading to undefined behaviour. As my method stands at the moment, it is up to the user to check whether such a situation can actually arise. With these two features, the interaction can happen, in the following scenario. Assume that subscriber A has GR, and B and C are in the ‘group’. If, for example B subscribes to TCS and has A on its screening list, then a call to A will result in an i\_alert from A to B, which B will answer with (o\_inform,B,A,“screened”). At this point A will be in a state introduced by Group Ringing, and will not be able to deal with that message.

Roughly 90% of the interactions I detected were discovered by applying these simple criteria. Yet, the interesting interactions are those that are not detected by these tests. The way that the contest model was set up, there were very few of these ‘tricky’ interactions, since most features clashed quite badly on the simple criteria. Assuming a good software engineering approach, though, which would take account of the ‘easy’ interactions and

aim to avoid them in the first place, the ‘tricky’ ones, i.e. those not detected by our syntactic criteria, become crucial. At this point, simulation, testing and model checking come into their own again, because the interactions that cannot easily be detected by syntactic criteria are likely to show up only in longer runs of the system.

## 4.7 Modelling POTS in CSP

### 4.7.1 A network of basic call processes

Modelling a single basic call in CSP is very simple: the states become CSP processes and all messages become events, the send and receive operations fit very nicely. A transition with multiple messages is split up into a sequence of events.

The problems start, however, when one composes several such basic call processes into a network. Now, the different processes may ‘de-synchronise’ since there is nothing to enforce the atomicity of the transitions in the diagram. Hence one line could embark on a transition if this was triggered by a local event and thereby prevent another line from sending a message to it. In other words, the naive, literal translation from the contest instructions allowed some internal choice between transitions that need to synchronise with other transitions, leading to possible deadlocks.

It turned out, though, that this problem was not too hard to solve: it was necessary to reorder the events labelling each transition, so that all external communication events came first on the respective transitions. Eventually, all transitions started with one input or output event, if they contained one at all, followed by only internal (local) events.<sup>13</sup> In CSP terms this meant offering all events that the line processes had to synchronise on as external choices.

Figure 4.3 shows a small fragment of the CSP model that I developed:  $\text{BCM}(state, x, y)$  represents the basic call state  $state$  for subscriber  $x$ , where  $y$  contains the telephone that  $x$  is connected to, if appropriate, otherwise  $y = x$ .

### 4.7.2 A specialised feature construct

To automate feature integration I extended the  $\text{CSP}_M$  syntax with a simplified, and somewhat specialised version of the feature construct, suited to the

---

<sup>13</sup>For some features this meant introducing extra states and transitions, since some of them used multiple send operations in one transition.

## 4.7. Modelling POTS in CSP

---

```
BCM(o_busyp, x, y) =
  i.m_free!y!x!none -> ring_tone!x -> BCM(o_wait_answer,x,y)
  [] i.m_busy!y!x!none -> busy_tone!x -> BCM(o_wait_onhook,x,y)
  [] o.m_stop_alert!x!y!none -> onhook!x ->
    bill.b_onhook!x!x!time -> BCM(idle,x,x)
  [] i.m_alert?z:Lx({x})!x?p -> o.m_busy!x!z.none ->
    BCM(o_busyp, x, y)

BCM(o_connected, x, y) =
  i.m_disconnect!y!x!none -> disconnect_tone!x ->
    BCM(o_wait_onhook,x,y)
  [] o.m_disconnect!x!y!none -> onhook!x ->
    bill.b_stop_o!x!y!time -> bill.b_onhook!x!x!time ->
    BCM(idle,x,x)
  [] i.m_alert?z:Lx({x})!x?p -> o.m_busy!x!z.none ->
    BCM(o_connected, x, y)
```

$Lx(\{x\})$  is the set of subscribers except for  $x$ .

Figure 4.3: CSP code for POTS states ‘o\_busy?’ and ‘o\_connected’

textual representation of the feature diagrams. We will denote the extended language by  $CSP_M^{FC}$ . Figure 4.4 shows the  $CSP_M^{FC}$  code for the Terminating Call Screening feature as an example.

The first simplification was that no `require` section was needed, since all features were going to be integrated only into one fixed base system or extensions (featured versions) thereof. Thus each feature required only a subset of what the base system (POTS) provided.

Secondly, the features used only one special form overriding, hence I dropped the keywords `change`, `override` and `introduce` in favour of a more compact notation.

Thirdly, the base system had quite a rigid syntactic structure due to the way it was derived from the transition diagram (Figure 4.1), which I exploited in writing the feature integrator.

After describing the specialised feature construct, I will outline how this relates to the general feature construct described in section 4.4.

In the specialised feature construct, a feature definition may contain any number of standard  $CSP_M$  definitions. These are treated as part of the `introduce` section and will simply be added to to the base system, and the user is responsible for avoiding name clashes (re-definitions).

The `change` clauses take the following form:

- Transitions can be added or replaced by means of special definitions with the following syntax:

```

-----
-- Feature Terminating Call Screening
-----
-- This is designed for a maximum of four phones: any phone above C
-- gets $TCSSetD as its screening set (only relevant if in subscriber
-- set $TCSSubs).
-----
$TCSSubs = {C}
$TCSSetA = {}
$TCSSetB = {}
$TCSSetC = {A}
$TCSSetD = {}

screen(A) = $TCSSetA
screen(B) = $TCSSetB
screen(C) = $TCSSetC
screen(_) = $TCSSetD

BCM(state:{idle},x,x) += if member(x,$TCSSubs)
                        then i.m_alert?y:Lx({x})!x!none -> TCS1(x,y)
                        else BCM(idle,x,x)

TCS1(x,y) = (if member(y,screen(x))
             then o.m_inform.x.y.screened -> BCM(idle,x,x)
             else BCM(t_busys,x,y))
            [] i.m_alert?z:Lx(x)!x!none -> o.m_busy.x.z.none -> TCS1(x,y)

BCM(state:{o_busyp},x,y) += i.m_inform.y.x.screened -> TCS2(x,y)

TCS2(x,y) = announce.x.tcs -> BCM(o_wait_onhook,x,y)
            [] i.m_alert?z:Lx(x)!x!none -> o.m_busy.x.z.none -> TCS2(x,y)

-- end of feature -----

```

`BCM(state,subscriber,partner)` denotes a basic call state,  
`$TCSSubs` is the set of subscribers to TCS (a parameter of the feature),  
`$TCSSetA` through `$TCSSetD` represent the screening lists of the respective subscribers (feature parameters).

Figure 4.4:  $CSP_M^{FC}$  code for Terminating Call Screening

*process*( $p_1, \dots, p_n$ ) += *new definition*

where *process* is a process name from the base file. The actual parameters can be ‘captured’ and are referred to by  $p_1, \dots, p_n$  in the *new definition*. It is also possible to restrict the range of parameters in certain cases. The *new definition* is the CSP code for the transition to be added.

The matching of the formal parameters (in the feature definition) to those in the base system is nothing special, although I did not go into it for the general feature construct.

In addition to these modifications of the feature construct, I allowed for parameters to a feature. In my implementation, any name prefixed with `$` is interpreted as a feature variable. Such variables can be assigned de-

fault values in the feature file which may be overridden on the command line when invoking the integrator (see below). All (non-defining) references in the feature file are textually replaced by the value thus given. In the abstract treatment of the general CSP feature construct this was irrelevant, for practical use however, having parametrised features is very helpful.

I used one special variable `$DEFAULT` as a placeholder for the code of the original transition. This greatly facilitated writing some of the features (eight out of the twelve in the contest). However, the effect of `$DEFAULT` cannot be emulated by the keyword `original` of general CSP feature construct, as I will explain below.

### Emulating the specialised feature construct

The important difference between the specialised and the general feature construct lies in the fact that the new feature construct allows to *add or replace* transitions, i.e. in a sense the change is conditional on the base system itself, not on some runtime conditions. If we want to bring the semantics of this new feature construct in line with the operational semantics defined in the previous section, we have to give a translation from the specialised feature construct into the general one. This can be solved in the following way.

Then we split each feature into two stages. The first stage adds a transition with the appropriate guard to the choice of transitions in a state, and allows non-determinism between the old and new transitions. In fact, in this step, only the guard matters; we could add mere stubs for the new transitions, e.g. in the form *event*  $\rightarrow$  *Stop*.

Potential unwanted transitions are eliminated in a second stage, using the `after` keyword: `after` the event in question we add the code that the feature specifies. This way, even if the guard we added was the same as one on an existing branch, the two would become identical, both containing the feature code rather than the base system code. Effectively, any transition of the base system state with the same guard as the feature transition gets overwritten.

For adding a completely new transition, this process works fine too: after the first stage, there is exactly one transition to which the `after` modifier refers; and this transition then is modified to contain the feature code.

There is however one drawback to this translation: we cannot keep the POTS code for basic call states parametrised by subscriber, as in Figure 4.3, while at the same time selecting which phones get which features. In the specialised feature construct the `$DEFAULT` placeholder could be used in an if-then-else construct, as for example in the feature Call Forward on Busy:

```
BCM(state:$CFBstates,x,y) +=  
  if member(x,CFBSubs)  
  then i.m_alert?z:Lx({x})!x?p -> CFB1(state,x,y,z)  
  else $DEFAULT
```

The resulting CSP processes test at runtime whether the feature's transition or the original transition should be available, depending on whether the process represented a featured telephone (`member(x,CFBSubs)=true`), or an unfeatured one. The translation presented here cannot provide this, since the code of the old transition is not available in the second stage of the integration process I described above. The only correct solution to this problem involves duplicating the basic call definitions in the base system, so that there is one for each subscriber. Then we can integrate the each feature into exactly those call processes which we want to endow with the feature, and retaining a copy of the original transition is not necessary.

### 4.7.3 Feature integration

The feature integrator was written in Python. It combines a feature definition (written in  $CSP_M^{FC}$ ) and a base system in  $CSP_M$  and produces a new  $CSP_M$  file defining the featured system. Features may be parametrised and the feature integrator allows to set values for the parameters at integrate time overriding any default values given in the feature definition are used.

The prototype I wrote for the contest relies quite heavily on syntactic conventions in our POTS model, e.g. += definitions only work for BCM states and the number of parameters is fixed. On the other hand this enabled us to substitute sensible defaults for the 'don't care' place holder (`_`).

After integrating a feature, the resulting  $CSP_M$  file can be used as the base file for further feature integrations or, of course, be analysed using PROBE and FDR2. Figure 4.5 shows the two basic call states 'idle' and 'o\_busy?' after integration of the feature TCS.

### 4.7.4 Detecting interactions

To detect feature interactions in the  $CSP_M$  model, I applied several techniques.

First I used FDR2 to check the featured systems for deadlock. Obviously the telephone system should never deadlock – it must always be possible for every line to get back to the initial state. However, since I was working on a system with four phones, I could not detect local deadlocks, i.e. situations in which one or more phones had no more enabled transitions but where

## 4.7. Modelling POTS in CSP

---

```
BCM(idle, x, _, p) =
  offhook!x -> bill.b_offhook!x!x!time ->
    BCM(o_offhook,x,x,none)
  [] (if member(x,TCSSubs)
    then i.m_alert?y:Lx({x})!x?q -> TCS1(x,y,q)
    else i.m_alert?y:Lx({x})!x?p -> BCM(t_busys,x,y,p))
  [] i?m:NoAlert(x) -> BC(x)

BCM(o_busyp, x, y, p) =
  i.m_free!y!x!none -> ring_tone!x -> BCM(o_wait_answer,x,y,none)
  [] i.m_busy!y!x!none -> busy_tone!x -> BCM(o_wait_onhook,x,y,none)
  [] o.m_stop_alert!x!y!none -> onhook!x ->
    bill.b_onhook!x!x!time -> BC(x)
  [] i.m_alert?z:Lx({x})!x?p -> o.m_busy!x!z.none ->
    BCM(o_busyp, x, y, none)
  [] i.m_inform.y.x.screened -> TCS2(x,y)
```

Figure 4.5: States ‘idle’ and ‘o\_busy?’ after integrating TCS

there was at least one which could still move – even if that only meant going offhook and onhook repeatedly. If FDR2 allowed the user to impose fairness constraints, such situations could be detected.

Secondly, I explored the behaviour of the system using PROBE, to test if the featured system *could* behave in the intended way. PROBE allows the user at every step to choose one of the enabled events and thus to simulate a run of the system. Hence it cannot be used to verify the absence of undesirable behaviour.

This is where FDR2 comes in again. While the previous two techniques are mainly for debugging, FDR2 can explore all possible executions of a system. The central method used for this is refinement checking in one of three models.<sup>14</sup> However, since features both add and remove behaviours, refinement is not such a useful notion.

Instead I coded desirable or undesirable patterns of behaviour as observer processes and composed them with the system in question, synchronising on the events that were relevant for the behaviour I wanted to test for. The observer processes were designed in such a way that the presence of the behaviours they represented lead to a deadlock in the composite system. This allowed for exhaustive checking of properties.

---

<sup>14</sup>Traces, failures and failures-divergence model.

### 4.7.5 Limitations

The expressive power of observer processes in CSP is rather weak. Unlike ‘never claims’ in SPIN/PROMELA, which uses Büchi automata to deal with infinite behaviours (e.g. liveness properties, in general recognition of  $\omega$ -regular traces), observer processes in CSP can only be used to detect the presence of finite traces.

As with SPIN, some observer processes led to a huge blow-up in the state space, which made it impossible to verify the properties.<sup>15</sup>

Another drawback of using observer processes is that they need to be coded by hand which is a rather error-prone procedure. Contrast this with expressing a property in temporal logic, with subsequent automatic translation to the corresponding Büchi automaton.

This list of drawbacks, together with the success I had with static analysis, might give the impression that the model-checking approach is deeply flawed. However, I would like to point out, that the fact that a simple static analysis detected such a large proportion of the feature interactions hinged on the specific model the contest defined. Also, the expressiveness of model-checking languages varies greatly, with regards to both the description of models and the properties that can be checked. Indeed, I would argue that model checking is still an invaluable tool in proving the absence of unwanted behaviour, and in finding deep errors.

## 4.8 Feature interactions

A full list of the (two-way) interactions I detected among the twelve features of the contest, with explanations can be found in [65]. Table 4.8 may give an impression of the sheer number of interactions that we found. I only elaborate on a small selection here.

- **TCS & GR:** Both features are triggered by an incoming alert message in the idle state (type 1 interaction). Therefore they cannot both be active on the same line.
- **TCS & CW:** When the a subscriber of Call Waiting is in a call, further incoming calls (i\_alert message) are not screened, since TCS is triggered only in the idle state; this is a type 2 interaction.
- **TCS & TWC:** A typical type 3 interaction occurs because Terminating Call Screening introduces a new message (i\_inform). All lines in

---

<sup>15</sup>... at least with the computing resources available to us.

Table 4.1: Feature Interactions Phase 1

	CFB	CNDB	CW	RBWF	RC	SB	TCS	TL	TWC	VM	CT	GR
CFB	××	–	–	–	–	–	–	–	–	–	–	–
CNDB	××		–	–	–	–	–	–	–	–	–	–
CW	××	×	×	–	–	–	–	–	–	–	–	–
RBWF	×××××	××	×××	×	–	–	–	–	–	–	–	–
RC	××			××		–	–	–	–	–	–	–
SB	××			××	×		–	–	–	–	–	–
TCS	×	×	×	×××				–	–	–	–	–
TL	×			××				×	–	–	–	–
TWC	×××	×	×××	××××	×	×	×	×		–	–	–
VM	×		××	××	×	×	×		××	×	–	–
CT	××××	×	×××	××××	×	×	××	××	×××	××	××	–
GR	××	×	×××	×××	×	×	××		××	××	××	×

Each × in the table stands for a distinct interaction.

We did not distinguish feature combinations by the order of feature integration, hence the top right half of the table is not used.

the network are upgraded so that they can deal with this new message, but only in BC states. So if someone uses Three Way Calling to call a Call Screening subscriber, an ‘i\_inform’ message from that line to the (TWC) caller cannot be processed by the TWC feature.

- **RBWF & RC:** Ring Back When Free initiates the ring-back without a dial message, therefore Reverse Charging will not be triggered. This is a type 4 interaction, however, if the Ring Back feature were redefined to use the dial message, we would still get an interaction, namely of type 2, because the dial message would not be issued from a basic call state, and consequently, RC would not be triggered.

## 4.9 Conclusions

In this chapter I have developed the syntax and semantics for a feature construct for CSP, which allows the user to define alterations to the way processes communicate with each other. It also enables the user to change the way processes are composed together, including the addition of new components, by means of overriding previous definitions.

This feature construct satisfies the criteria for feature constructs that I listed in chapter 1, in that it makes features self-contained, easy to specify and easy to add and remove from a system and easy to revise. Others may judge if it also makes features easy to understand. For this feature construct I have given operational semantics to fit in with the operational semantics for CSP defined by Roscoe [67] and Scattergood[68].

Sadly there was no time to conduct a full case study with this feature construct, as the participation in the Feature Interaction Contest took precedence. For this contest I devised a different, somewhat simpler feature construct, which was well adapted to the way that the features of the contest were specified. However, I have shown that this specialised feature construct can be emulated using the general one, without sacrificing too much expressiveness.

The Feature Interaction Contest involved modelling a telephone system with a detailed messaging protocol and defining twelve features of varying complexity, which were then tested for pairwise and three-way interactions. The verification task was made difficult by the lack of a clear description of what was considered incorrect or undesirable behaviour on the one hand, by certain shortcomings of the model checker FDR2 on the other. The former is probably quite realistic, because at specification time, the requirements are often not fixed, and only in the process of “playing with the system” does one

## 4.9. Conclusions

---

discover contradictory requirements, or additional assumptions that need to hold.

In my work on the Feature Interaction Contest I found a good technique detecting potential feature interactions without having to specify the features in  $CSP_M^{FC}$  and integrating them. The success of this syntactic method may seem to cast doubt on the value of model checking for the detection of feature interactions, but I would not have reached an ‘implementation’ of POTS and the features without the aid of model checking. If PROBE and FDR2 had been nothing more than debugging aids in the development of my system, they would have been invaluable in getting the system right, and moreover in gaining a good understanding of it.

Furthermore, I believe that the syntactic method captures mainly the ‘obvious’ interactions. Once these are out of the way, one needs a method to find those interactions that can result, e.g. from different interpretations of the same data in different features, or from the variables not being reset after a call. These interactions may result in strange behaviour or simply in the violation of invariants, but this will only become visible in runs of the system.

My experiences with FDR2 as a model checker were mixed. While FDR2 was very efficient in proving the absence or presence of deadlocks in the telephone system, no matter whether one, two or three features had been added, it lacks an expressive property language, such as LTL or CTL. Trying to make up for this loss by coding observer processes did not always yield satisfactory results: often the verification task then took FDR2 to its limits (or rather to the limits of the hardware available). Perhaps a future version of FDR2 will offer LTL model checking like SPIN does.

# Chapter 5

## Conclusions

### 5.1 Specify features with feature constructs

In the last three chapters we have seen feature constructs for three quite different specification languages. The basic idea was the same, though: to offer a way of specifying features as separate entities, with as little reference to the details of the base system specification as possible. One has to devise a feature construct for each language, to fit both syntactically and semantically. In fact, as we have seen for CSP, there may even be different feature constructs for different purposes<sup>1</sup>. If the class of base systems is syntactically constrained, as was the case for the POTS model for the Feature Interaction Contest, it makes sense to use a specialised feature construct because it

- can be more concise,
- can be more expressive since one can exploit additional knowledge about the system with and without features,
- makes it easier to implement a feature integrator.

On the other hand, in some cases a specialised feature construct will limit the range of features that can be specified. However, this may very well coincide with the restrictions on the class of systems investigated. For example, the model of the telephone system for the Feature Interaction Contest stipulated a structure which made it very difficult to implement (on the model level) certain features.

---

<sup>1</sup>Note that aspect-oriented programming likewise may use different aspect languages for different ‘aspects’ or concerns [50].

## 5.2 The usefulness of feature constructs

Feature constructs have proved to be helpful for specifying features, for understanding feature functionality, and for revising features. No matter whether we are talking about specifications or actual code, without a feature construct, adding a feature to a system means altering various parts of the system. This makes it inherently difficult to understand what a feature does, and even harder to change a feature’s functionality. And for much the same reason, specifying a feature becomes easier: the specifier writes a more or less self-contained specification, and a feature integrator performs the appropriate alterations of the base system. Since a feature construct separates the basic functionality from optional add-ons, i.e. features, it makes it easy to tailor different versions of a system for different purposes.

Not only in this sense do feature constructs address the feature interaction problem, especially when taken at specification level. By giving clear semantics to features, a feature construct also facilitates reasoning about combinations of features, in some cases theoretical results about feature semantics may even eliminate the need to test for certain interactions (*cf.* Theorem 2.5.25).

Currently feature-oriented specification suffers from a lack of tool support. Various groups are working on tools and methods, but the widely used specification languages and validation and verification methods have so far not been extended to allow for a feature-oriented approach.

## 5.3 Benefits and limits of model checking

Model checking has seen a boom in recent years. This is especially true for hardware design, but it has also entered software design beyond protocol design (*cf.* [39]). For example, the SDL<sup>2</sup> tool SDT (now superseded the Telelogic Tau<sup>(TM)</sup> tool set) includes a model checker for SDL specifications. The principal reason for the success of model checking is that it is “push button technology”: it offers proof of correctness (w.r.t. specified properties or in terms of refinement) without the need to prove theorems, which would require user interaction and a firm grounding in mathematics. Furthermore, the convenience of model checking lets us go through the cycle of specifying, testing and revising a lot quicker than methods involving theorem proving.

However, the state space explosion problem means that model checking does not scale up to real life systems, particularly in software engineering.

---

<sup>2</sup>Specification and Description Language

One can generally expect to verify only parts of the system under investigation, or an abstraction.

Even so, being able to experiment with a specification or program by asking whether it has certain properties or can perform certain behaviours is an invaluable help in understanding the system. This goes beyond what simulation can offer: in simulation the developer can try only a small number of behaviours, maybe enough to see that in the right circumstances the system behaves as expected. Model checking explores all possible behaviours, including ones that no-one anticipates.

Here lies another great advantage of model checking: when the model checker finds a violation of a property, it usually illustrates it with an example trace, helping the specifier to pinpoint the cause of the problem. Furthermore, model checking can uncover “deep” problems, which are hard to find using traditional analysis techniques (static analysis, simulation, abstract interpretation).

There will always be a tradeoff between the coarseness and the complexity of the analysis: model checking is highly complex and poses enormous demands in terms of time and memory required, while e.g. simulation requires only little time and memory, but is far less likely to find all errors.

Improvements in the efficiency of model checking seem rather small in the light of the state explosion problem. New representations and algorithms only yield constant factors in terms of time and memory requirements, and the increase in computing power cannot compensate for the state explosion because the complexity of systems we would like to analyse grows at a similar rate. However, advances in the use of abstraction techniques and compositional methods can increase the size of systems amenable to model checking, as is demonstrated, for example, by Cadence SMV<sup>3</sup>.

Another area in which model checking has shortcomings is the expressiveness of the specification languages and the logics used. Current model checking languages are very far removed from programming languages, not only in the fact that they are designed to describe only finite state systems, but also in the richness of the language. To my knowledge, currently no model checker offers dynamic data structures or multiple models of communication, with the exception of FDR2 and the SDT Validator. And even these tools do not cope very well with such data structures. While it is possible in most languages to simulate these, this greatly increases the complexity of the system to be analysed. The other side of the coin is the property language or logic used. Especially in the case of the telephone network, many prop-

---

<sup>3</sup>Cadence SMV and related publications are available from <http://www-cad.eecs.berkeley.edu/~kenmcmil/>

erties require very involved formulas to describe approximately the desired behaviour. By refining the logics, it would be possible to give more precise descriptions of properties using simpler formulas. One would like to be able to express properties in terms of capabilities and strategies. For example, in ATL (Alternating-Time temporal Logic), one can formulate properties like: “The telephone switch can ensure that a certain user cannot be called.”

## 5.4 Directions for further research

For obvious reasons, this thesis could not cover everything relating to the specification of features. A lot of theoretical work remains to be done, let alone implementation of tools which exploit these results.

I did not implement feature integrators for the PROMELA feature construct or the general CSP feature construct. The former because of problems with the restrictions of PROMELA’s grammar, which make it all but impossible to automatically integrate a feature so that a second integration will still be possible. The CSP feature integrator could not be realized because the Feature Interaction Contest took precedence, which meant that we needed a working solution rather quickly. Furthermore, the contest model including the features was far easier to implement with a specialised feature construct and integrator.

Mark Ryan and I have defined the semantics of our SMV feature construct and we have been able to show some results as to when features for SMV programs do not interact. With the semantics of the CSP (section 4.4.2) I have laid a basis for similar work for the CSP feature construct. However, I did not have time to prove any theorems.

On a more abstract level, too, some more questions have arisen that my studies could not cover. One direction to make model checking of featured systems more tractable would be to investigate “feature-led” abstractions, i.e. simplifications of a complex featured system which leave the changes that the features introduce exposed, while “abstracting away” as much of the system as possible.

In a similar vein, proof by induction and related techniques could help to reduce the size of systems: if it can be shown that there is a minimal system which displays all possible feature interactions (up to permutation or duplication), one would not need to check any larger systems. By way of illustration, in the telephone system all subscribers using the same set of features (with the same parameters) are equivalent. Hence, if we can show that a configuration of, say, three subscribers leads to an interaction, that interaction will occur for any configuration containing equivalent subscribers.

The most difficult part of such a method will be to establish a minimum set *a priori*.

In [17] Cassez *et al.* prove non-interaction results for certain classes of features, using alternating-time temporal logic (ATL) to specify desirable properties of a system. ATL allows the specifier to talk about capabilities of different parts of the system, e.g. different users, the switch, or a feature. This gives the opportunity for a much more detailed description (in the logic) of the properties one is interested in. Hence, with ATL as the property language, one can prove stronger or at least more relevant non-interaction results.

# Appendix A

## List of abbreviations

ACB	Automatic Call Back – this feature records the number of the last caller to the subscriber’s phone, which the subscriber can choose to ring directly, without dialling the number
ATL	Alternating-Time temporal Logic [2]
BT	British Telecom
CCS	Robin Milner’s process Calculus of Communicating Systems [57]
CFB	Call Forward on Busy – divert calls when the line is busy
CFNR	Call Forward on No Reply – all calls to the subscriber’s phone which are not answered after a certain amount of time, are diverted to another phone.
CFU	Call Forward Unconditional – all calls to the subscriber’s phone are diverted to another phone
CNDB	Calling Number Delivery Blocking – do not display the callers number to the callee
CSP, $CSP_M$	C.A.R. Hoare’s process calculus (Communicating Sequential Processes [38]), machine readable CSP
CT	Call Transfer – transfer an active call to another telephone
CTL, CTL*	Computation Tree Logic
CW	Call Waiting – allow subscriber to take a second call and switch between first and second call

---

FDR, FDR2	a model checker for CSP (“FDR” stands for “Failures, Divergence, Refinement”)
FSM	Finite State Machine
GR	Group Ringing – make three telephones ring for an incoming call to one of them, stop the ringing when one of them is answered
IN	Intelligent Network (ITU Recommendations Q.1200 series [43])
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization ( <a href="http://www.iso.ch/">http://www.iso.ch/</a> )
ITU	International Telecommunication Union ( <a href="http://www.itu.int/">http://www.itu.int/</a> )
LOTOS	Language Of Temporal Ordering Specification (ISO 8807 [42])
LTL	Linear Time temporal Logic
LTS	Labelled Transition System
OCS	Originating Call Screening – this feature inhibits calls from the subscriber’s phone to any number from a set chosen by the subscriber. Any attempt to ring such a number will yield an announcement.
POTS	Plain Old Telephone System
PROMELA	“Protocol Modelling Language”, the input language for SPIN
RBWF	Ring Back When Free – ring back callers that got the busy tone (also known as Automatic Call Back)
RC	Reverse Charging – callee pays for call
SB	Split Billing – callee pays part of the call
SDL	Specification and Description Language (ITU Recommendation Z.100 [43])
SMV	Symbolic Model Verification system [55]
SPIN	(not an acronym) model checking system for the language PROMELA and LTL [39] ( <a href="http://netlib.bell-labs.com/netlib/spin/whatispin.html">http://netlib.bell-labs.com/netlib/spin/whatispin.html</a> )

---

TCS	Terminating Call Screening – block calls from certain phones to the subscriber’s phone
TL	TeenLine – a PIN must be entered before calls can be made
TWC	Three-Way Calling – allow the subscriber to initiate a second call and let all three parties talk with each other (also known as Conference Call)
VM	Voice Mail – callers can leave a message if the phone is not answered (also known as CallMinder)

# Bibliography

- [1] R. Accorsi, C. Areces, W. Bouma, and M. de Rijke. Features as constraints. In Magill and Calder [53].
- [2] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Lecture Notes in Computer Science*, 1536:23–60, 1998. available from [http://www-cad.eecs.berkeley.edu/~tah/Publications/alternating-time\\_temporal\\_logic.ps](http://www-cad.eecs.berkeley.edu/~tah/Publications/alternating-time_temporal_logic.ps).
- [3] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In Magill and Calder [53].
- [4] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and LOTOS. In Magill and Calder [53].
- [5] G. C. Barney and S. M. dos Santos. *Elevator Analysis, Design and Control*. IEE Control Engineering Series 2. Peter Peregrinus Ltd., 1985.
- [6] M. Berry. Proving properties of the lift system. Master’s thesis, School of Computer Science, University of Birmingham, 1996.
- [7] J. Blom, R. Bol, and L. Kempe. Automatic detection of feature interactions in temporal logic. In Cheng and Ohta [19], pages 1–19.
- [8] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In Bouma and Velthuisen [9], pages 197–216.
- [9] L. G. Bouma and Hugo Velthuisen, editors. *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands, May 1994. IOS Press.
- [10] J. Brederke. Families of formal requirements in telephone switching. In Magill and Calder [53].

## BIBLIOGRAPHY

---

- [11] G. Bruns, P. Mataga, and I. Sutherland. Features as service transformers. In Kimbler and Bouma [51], pages 85–97.
- [12] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Feature interaction visualization and resolution in an agent environment. In Kimbler and Bouma [51], pages 135–149.
- [13] M. Calder and A. Miller. Analysing a basic call protocol using PROMELA/XSPIN. Technical report, Department of Computing Science, University of Glasgow, 1998.
- [14] E. J. Cameron, N. Griffeth, Y.-J. Lin and M. E. Nilson, W. K. Schnure, and H. Velthuisen. A feature interaction benchmark for IN and beyond. In Bouma and Velthuisen [9], pages 1–23.
- [15] C. Capellmann, P. Combes, J. Petterson, B. Renard, and J. L. Rutz. Consistent interaction detection – A comprehensive approach integrated with service creation. In Dini et al. [23], pages 183–197.
- [16] Franck Cassez. A model to describe feature integration. submitted, April 1998.
- [17] Franck Cassez, Mark Dermot Ryan, and Pierre-Yves Schobbens. Proving feature non-interaction with Alternating-Time Temporal Logic. In Gilmore and Ryan [29], pages 85–104.
- [18] T. Charnois. A natural language processing approach for avoidance of feature interactions. In Dini et al. [23], pages 347–363.
- [19] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications III*, Tokyo, Japan, October 1995. IOS Press.
- [20] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a re-implementation of SMV. Technical Report 9801-06, IRST, Trento, Italy, January 1998.
- [21] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, volume 684, pages 124–175. Springer, 1994.
- [22] Douglas D. Dankel II, Mark Schmalz, Wayne Walker, Karsten Nielsen, Luiz Muzzi, and David Rhodes. An architecture for defining features and exploring interactions. In Bouma and Velthuisen [9], pages 258–271.

- [23] P. Dini et al., editors. *Feature Interactions in Telecommunications and Distributed Systems IV*, Montreal, Canada, June 1997. IOS Press.
- [24] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation: a synchronous point of view. In Kimbler and Bouma [51], pages 262–275.
- [25] A. P. Felty and K. S. Namjoshi. Feature specification and automatic conflict detection. In Magill and Calder [53].
- [26] Formal Systems (Europe) Ltd, Oxford, UK. *Failures-Divergence Refinement*, October 1997.
- [27] A. Gammelgaard and J. E. Kristensen. Interaction detection, a logical approach. In Bouma and Velthuijsen [9], pages 178–196.
- [28] P. Gibson, G. Hamilton, and D. Méry. A taxonomy for triggered interactions using fair object semantics. In Magill and Calder [53].
- [29] Stephen Gilmore and Mark Ryan, editors. *Language Constructs for Describing Features*. Springer-Verlag London Ltd, 2000/2001.
- [30] R. Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes, 1987.
- [31] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta. Feature interaction detection contest. In Kimbler and Bouma [51], pages 327–359.
- [32] N. D. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In Bouma and Velthuijsen [9], pages 217–235.
- [33] Nancy Griffeth, editor. *1st International Workshop on Feature Interactions in Telecommunications Software Systems*, St. Petersburg, Florida, USA, December 1992.
- [34] R. J. Hall. Feature interactions in electronic mail. In Magill and Calder [53].
- [35] Robert J. Hall. Feature Combination and Interaction Detection via Foreground/Background Models. In Kimbler and Bouma [51], pages 232–246.
- [36] M. Heisel and J. Souquière. A heuristic approach to detect feature interactions in requirements. In Kimbler and Bouma [51], pages 165–171.

## BIBLIOGRAPHY

---

- [37] T. A. Henzinger. The theory of hybrid automata. *IEEE*, (1043-6871/96):278–292, 1996.
- [38] C. A. R. Hoare. *Communication Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
- [39] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. available from <http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html>.
- [40] G. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. available from <http://cm.bell-labs.com/cm/cs/who/gerard/gz/ieee97.ps.gz>.
- [41] M. R. Huth and M. D. Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 1998.
- [42] International Organization for Standardization. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour – ISO 8807*, 1989.
- [43] ITU-T. *Specification and Description Language – ITU Recommendation Z.100*, 1980–1992.
- [44] B. Jonsson, T. Margaria, G. Naeser, J. Nyström, and B. Steffen. Incremental requirement specification for evolving systems. In Magill and Calder [53].
- [45] J. Kamoun and L. Logrippo. Goal-oriented feature interaction detection in the intelligent network model. In Kimbler and Bouma [51], pages 172–186.
- [46] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
- [47] Y. Kawarasaki and T. Ohta. A new proposal for feature interaction detection and elimination. In Cheng and Ohta [19], pages 127–139.
- [48] Dirk O. Keck. A tool for the identification of interaction-prone call scenarios. In Kimbler and Bouma [51], pages 276–290.
- [49] A. Khoumsi and R. J. Bevelo. A detection method developed after a thorough study of the contest held in 1998. In Magill and Calder [53].

- [50] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer, June 1997.
- [51] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*, Lund, Sweden, September 1998. IOS Press.
- [52] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Second feature interaction contest. In Magill and Calder [53]. Available from <http://www.comms.eee.strath.ac.uk/~fi/fiw00/contest-instr2.ps>.
- [53] E. Magill and M. Calder, editors. *Feature Interactions in Telecommunications and Software Systems VI*, Glasgow, Scotland, UK, May 2000. IOS Press.
- [54] D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In Kimbler and Bouma [51], pages 115–134.
- [55] K. L. McMillan. *Symbolic model checking*. Kluwer Academic, 1993.
- [56] K. L. McMillan. The SMV language. Available from <http://www-cad.eecs.berkeley.edu/~kenmcmil/>, March 1999.
- [57] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [58] M. Nakamura, Y. Kakuda, and Tohru Kikuno. Petri-net based detection method for non-deterministic feature interactions and its experimental evaluation. In Dini et al. [23], pages 138–152.
- [59] M. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo. Feature interaction filtering with use case maps at requirements stage. In Magill and Calder [53].
- [60] T. Ohta and Y. Harada. Classification, detection and resolution of service interactions in telecommunication services. In Bouma and Velthuisen [9], pages 60–72.
- [61] Y. Peng, F. Khendek, P. Grogono, and G. Butler. Feature interactions detection technique based on feature assumptions. In Kimbler and Bouma [51], pages 291–298.

## BIBLIOGRAPHY

---

- [62] M. Plath and M. Ryan. The feature construct for SMV: Semantics. In Magill and Calder [53].
- [63] M. C. Plath and M. D. Ryan. Plug and play features. In W. Bouma, editor, *Feature Interactions in Telecommunications Systems V*, pages 150–164. IOS Press, 1998.
- [64] M. C. Plath and M. D. Ryan. Feature integration using a feature construct. To appear in *Science of Computer Programming*, Elsevier Publishing, 1999.
- [65] M. C. Plath and M. D. Ryan. Entry for FIW'00 Feature Interaction Contest. Technical report, School of Computer Science, University of Birmingham, February 2000. Available from `ftp://ftp.cs.bham.ac.uk/pub/authors/M.D.Ryan/00-fiw-contest.ps.gz`.
- [66] S. Reiff. Identifying resolution choices for an online feature manager. In Magill and Calder [53].
- [67] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1999.
- [68] Bryan Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, The Queen's College, 1998.
- [69] C. Stirling. Modal and temporal logic. *Handbook of Logic in Computer Science*, volume 2.
- [70] M. Svensson and M. Andersson. Analysis of feature interactions in mobile terminals. In Kimbler and Bouma [51], pages 318–324.
- [71] M. Thomas. Modelling and analysing user views of telecommunications services. In Dini et al. [23], pages 168–182.
- [72] S. Tsang and E. Magill. Behaviour based run-time feature interaction detection and resolution approaches for intelligent networks. In Dini et al. [23], pages 254–270.
- [73] K. J. Turner. An architectural description of features and their interactions. submitted to *Computer Networks & ISDN Systems*, August 1997.
- [74] K. J. Turner. Validating architectural feature descriptions using LOTOS. In Kimbler and Bouma [51], pages 247–261.

- [75] Rob van der Linden. Using an architecture to help beat feature interaction. In Bouma and Velthuisen [9], pages 24–35.
- [76] H. Velthuisen. Issues of non-monotonicity in feature-interaction detection. In Cheng and Ohta [19], pages 31–42.
- [77] T. Yoneda and T. Ohta. A formal approach for definition and detection of feature interactions. In Kimbler and Bouma [51], pages 202–216.
- [78] R. Zygan-Maus. Feature interaction management for public switched networks. In Kimbler and Bouma [51], pages 32–44.