Plug-and-play features*

Malte Plath Mark Ryan
School of Computer Science
University of Birmingham
Birmingham B15 2TT
UK
http://www.cs.bham.ac.uk/{~mcp,~mdr}

May 7, 1998

Abstract

We propose a *feature construct* for defining features, and use it to provide a plug-and-play framework for exploring feature interactions. Our approach to the feature interaction problem has the following characteristics:

- Features are treated as first-class objects during the development phase;
- A method for integrating a feature into a system description is described. It allows features to override existing behaviour of the system being developed;
- A prototype tool has been developed for performing the integration;
- Our approach allows interactions between features to be witnessed.

In principle, our approach is quite general and need not be tied to any particular system description language. In this paper, however, we develop the approach in the context of the SMV model checking system.

We describe two case studies in detail: the lift system, and the telephone system.

1 Introduction

The concept of *feature* has emerged in telephone systems analysis as a way of describing optional services to which telephone users may subscribe. Features offered by telephone companies include Call Forwarding, Automatic Call Back, and Voice Mail. Features are not restricted to telephone systems, however. Any part or aspect of a specification which the user perceives as having a self-contained functional role is a feature. For example, a printer may exhibit such features as: ability to understand PostScript; Ethernet card; ability to print double-sided; having a serial interface; and others. The ability to think in terms of features is important to the user, who often understands a complex system as a basic system plus a number of features. It is also an increasingly common way of designing products.

Just as features are not restricted to telecommunication systems, the feature interaction problem can be observed in other contexts as well. To mention but a few examples, system extensions for Windows and Mac OS, packages for GNU Emacs and LaTeX styles may not

*Financial support from the EU through Esprit working groups ASPIRE (22704) and FIREworks (23531), and from British Telecom and the Nuffield Foundation in the UK is gratefully acknowledged.

work as intended when loaded in the wrong order, or in some cases not be compatible at all. These 'interactions' can usually be traced down to the fact that two 'features' manipulate the same entities in the base system, and in doing so violate some underlying assumptions about these entities that the other 'features' rely on. An example of interfering LaTeX styles are `german.sty` and `amstex.sty` (loaded in this order): when `amstex.sty` applies its changes, it is not aware of the alterations which `german.sty` has made, leading to undesirable results. In this case, luckily, reversing the loading order solves the problem, since `german.sty` was written to respect `amstex.sty`.

Feature interaction seems unavoidable as soon as the structure of data becomes flexible and the 'features' are not just filters on a one-way stream of data.[1] When a feature adds conceptually new information to a system or the data it works on, other features may be subverted. For example, if Call Waiting introduces a new state for which none of the other features have been prepared, their actions may not have the desired effect. But that is the central point of features: they may add functionality to a system which was not conceived when the system was designed. Thus feature interaction will occur in any sufficiently flexible system.

Since there is no way of avoiding feature interaction short of rigidly restricting the set of potential features, it is reasonable to analyse potential interactions as early as possible in the life-cycle of a new feature, and to interleave all steps in the development of new services with further analysis.

Our approach addresses the early stages of specification, and enables the specifier to identify problems with little more than the requirements to work from. That is to say, given a model of the basic system, the features are easy to specify, to add, and to remove or to re-specify, should interferences with other features arise.

We model the basic system and its features as different textual units, and *integrate* the features into the basic system, producing an extended system. We check for interactions by verifying the extended system. This approach works in principle with any modelling language and verification method. In this paper, however, we 'instantiate' the approach by working with the SMV model checker developed at Carnegie Mellon University [4, 7]. SMV can automatically check whether a system description satisfies its specification, expressed as a temporal logic formula. It does so by exhaustive state enumeration. A short introduction to SMV is provided in the appendix.

We have extended the SMV language with a new construct for describing features. We have built a tool called SFI ("SMV Feature Integrator") which compiles descriptions in this extended language into pure SMV, ready for verification by the SMV model checker. We present details of this extension and integration in the remainder of the paper, along with two substantial case studies of feature integration.

The structure of this paper is as follows: in the following section we describe the ideas behind our approach. This is followed by an explanation of our feature construct in section 3. Sections 4 and 5 are devoted to our two case studies, the lift system and the telephone system respectively. We conclude our paper by summing up our experiences with this approach in section 6.

There are three appendices. The first one, section A, is a a short introduction to the SMV system. Appendices B and C provide extra details of the case studies which space does not

[1] The structure of plug-in architectures for software usually enforces exactly this discipline: only one feature may manipulate the data at any one time, and the data has to be in a strictly defined format before and after.

permit to be included in the body of the paper.

2 Features and feature-integration

The general idea of our approach is to describe features formally as units of functionality which can be understood without detailed knowledge of the base system. These are then automatically integrated into the system, and the resulting extended system is verified. We do not assume any particular architecture of the base system in question, and (theoretically) as much or as little as one wants can be modelled. To make model checking viable, however, the system should be modelled in a rather abstract way, in order to keep the state space to a reasonable size. Since our approach aims at exposing logical interactions (e.g., inconsistencies), this is an advantage rather than a shortcoming, for at a high level of abstraction the logical interactions become more visible.

A feature description can be seen as a prescription for extending and changing the basic system. A feature description can usually be applied to different system descriptions, reflecting the fact that most features are quite generic, and only their implementations for different systems need to be adjusted to the precise underlying system.

The main aim of our approach of extending a specification and verification language with a feature construct is to provide a 'plug-and-play' system for experimenting with features and witnessing their interactions. Features can override existing behaviour of the base system in a tightly controlled way.

In this paper, we apply our approach to the SMV modelling language and verification tool [4, 7]. We extend the SMV language with a *feature construct*, thus making features self-contained textual units. These are integrated into the system description automatically by our tool, SFI ("SMV Feature Integrator"), and the resulting system can then be validated with the SMV model checker. We believe our approach is quite general, however. A similar tool could be developed for other description languages.

We chose SMV as the starting point for our approach for the following reasons:

- The SMV language is designed and optimised for concurrent, reactive systems, such as the telephone system.
- The SMV tool can check temporal properties of systems described using the SMV language. This enables rapid development of rigorous and accurate examples.

Our concept of feature makes it a special case of *superimposition* [6]. A superimposition is a syntactic device for adding extra code to a given program, usually to make it better behaved with respect to other concurrently running programs. In the classic example of superimposition, extra code is added to enable processes to respond to interrogations from a supervisory process about whether they are awaiting further input, and this enables smooth termination of the system. The superimposition construct proposed in [6] is suited to imperative languages, and therefore cannot be used directly for SMV.

3 The feature construct for SMV

In this section, we present an extension of the SMV syntax for describing features. We also show how model descriptions written in the extended SMV can be compiled into pure SMV,

```
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-clause | treat-clause ] }*
]
END
```

where: *impose-clause* stands for: `IMPOSE` *assignments*
treat-clause stands for: `TREAT` var_1 `=` $expr_1$ [`,` $\ldots var_n$ `=` $expr_n$]
[ ] stands for 'optional'
[| |] stands for 'one of'
{ }* stands for 'several'

Figure 1: The syntax of the feature construct

thus giving semantics to the feature construct. We will illustrate its use with some examples in the following two sections.

A formal specification of the syntax of the feature construct is given in figure 1. 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 old modules and variables that are used in the `INTRODUCE` and `CHANGE` sections should be `REQUIRE`d, 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, and CTL formulas in `SPEC` clauses may be given. These are textually added to the SMV text at integrate-time.

The `CHANGE` section specifies what the feature actually does. 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.

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

- It checks that the `REQUIRE`d entities are present in the base system, and reports an error if they are not.

- It inserts text for the new modules or variables declared in the `INTRODUCE` section.
- For `CHANGE`s 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.

- For `CHANGE`s of the form
  `IF` *cond* `THEN IMPOSE` x `:=` *expr*`;`
  In assignments x `:=` *oldexpr* or `next(`x`)` `:=` *oldexpr*, it 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.

- For `CHANGE`s that are not guarded by `IF` *cond* `THEN`, the `case` statements are of course omitted, and the variable, or respectively, the expression (x or *oldexpr*, respectively) are replaced directly by the new expression (*expr*).

The feature integration 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.

The semantics of `TREAT` and `IMPOSE` can also be given directly in terms of the automaton, rather than in terms of the SMV text. This is mainly of theoretical interest and we omit it for the sake of brevity.

4 Case study 1: the lift system

As a first case study, we have analysed the *lift system* and its features. For the base system we have adapted the lift system description written by Mark Berry [2]. The SMV code for a single lift travelling between 5 floors is given in the appendix (section B). It consists of about

120 lines of SMV code.

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 a landing button guarantees that the lift will arrive at that landing and open its doors.* In CTL[2]:

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

In the Appendix (section B) we list a further 6 properties of the basic lift system, and their CTL formulations.

Features of the lift system. The following features of the lift system were described using our 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 expected to be at the specified floor. The parking floor may be different at different times of the day, anticipating 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, we give the code for the parking feature in figure 2. The parking feature introduces the specification

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

which says that the lift will not remain idle indefinitely at any floor other than floor 1.

The other features mentioned introduce other specifications; these are listed in the appendix (section B).

Our method provides a framework to plug these different features into the lift system, and by examining the result, to witness feature interactions. Our SFI tool integrates one or more

[2]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 specification 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))
```

```
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: The code for the Parking feature

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 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 list just one ordering. (Thus, inspection of the table reveals that the only features which do not commute are Lift-$\frac{2}{3}$-full and Executive Floor.)

The properties, represented by columns in the table, are divided into two groups. 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 the properties which are designed to test the integration of specific features.

Feature(s) \ Property (see appendix B)	Landing button guarantuees service	Lift button guarantuees service	Lift doesn't change dir. while calls ahead	Lift door may remain closed	Lift may park at any floor	Lift may stop for landing calls (up)	Lift may stop for landing calls (down)	Lift travels empty only for landing calls	Lift btn. guarantees service unless empty	Doors will not close if overloaded	Lift will not move while overloaded	Lift will park at floor 1	Car calls take precedence when $\frac{2}{3}$ full	Btn. on exec. floor guarantuees service
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	×	×	√	√	×	√	√	—	—	—	—	—	×	×

Table 1: Feature interactions for the lift system

5 Case study 2: the telephone system

Our second case study 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 to indicate that her call is being held.

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

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 inhibits 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.

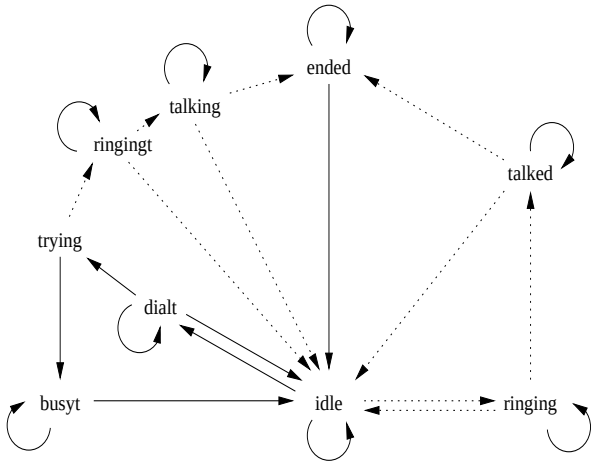
Automatic Call Back (ACB) This feature records the number of the last caller to the subscriber's phone, which the subscriber can choose to ring directly.

5.1 The base system (POTS)

We have built an SMV description of four synchronous phones. The behaviour of each phone is given by the finite automaton shown in figure 3, plus one variable, `dialled`, which indicates which other phone it is connected to (or trying to connect to). 'Idle' is the usual state of the phone; 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. 'Talking' represents the state in which 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 person we were talking to (or being talked to by) has hung up.

The dotted lines indicate that the transition must synchronise with a certain transition in a certain other phone. (For example, the transition from 'trying' to 'ringingt' must synchronise with the transition from 'idle' to 'ringing' in the phone we are trying to call.) The variable `dialled` determines which other copy of the phone automaton it has to synchronise with. 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 at non-deterministically. If a transition has to synchronise (indicated by a dotted line in the diagram) with a transition in another phone, it can only be chosen if the other phone chooses the

corresponding transition. A piece of the code for the phone module can be seen in figure 4. 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.



(Dotted lines indicate synchronising transitions.)

Figure 3: The automaton for a single phone.

As it turned out, this model quickly grew to large to verify when we added features, 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 3), the left hand side represents the originating line, and the right hand side the terminating line, both including the states 'idle' and 'ended', of course. Additionally each feature was only added to one of the phones. A positive side-effect of this differentiation is that it makes it easy to distinguish the interactions according to how features are distributed over the system. This goes some way towards the distinction between SUSC, SUMC, MUSC and MUMC (Single/Multiple User – Single/Multiple Component) interactions as introduced in [3], even at this rather high level of abstraction.

5.2 Integrating features into the telephone system

As an illustration of the feature construct we show the Ring Back When Free feature in figure 5. 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 don't 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 `dialled` and `st` are required, and the domain of `dialled` 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-number` and `rbwf-use`, and defines which number to store in `rbwf-number`, and when RBWF may be activated (`rbwf-use=1`) and deactivated (`rbwf-use=0`).

```
init(st) := idle;
next(st) :=
  case
    -- (...) --
    st=dialt : {idle, dialt, trying};
    st=trying :
      case
        dialled=2 & p[2].st=idle & next(p[2].st)=ringing
          & ((p[3].st=trying & p[3].dialled=2)->next(p[3].st)=busyt)
          & ((p[4].st=trying & p[4].dialled=2)->next(p[4].st)=busyt) :ringingt;
        dialled=3 & p[3].st=idle & next(p[3].st)=ringing
          & ((p[2].st=trying & p[2].dialled=3)->next(p[2].st)=busyt)
          & ((p[4].st=trying & p[4].dialled=3)->next(p[4].st)=busyt) :ringingt;
        dialled=4 & p[4].st=idle & next(p[4].st)=ringing
          & ((p[2].st=trying & p[2].dialled=4)->next(p[2].st)=busyt)
          & ((p[3].st=trying & p[3].dialled=4)->next(p[3].st)=busyt) :ringingt;
        1                                                            :busyt;
      esac;
    -- (...) --
  esac;
```

Figure 4: An extract from the code for the phone automaton

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 with the stored number.

We do 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. It would, however, slow down the model checking significantly, as we would have to introduce another variable to indicate the special ringing. In fact this could be implemented as another feature.

Properties. Some generic properties which we have verified for the (unfeatured) phone are given in section C. Apart from these generic properties, we also want to verify that the base system with the feature actually behaves as the feature specification demand. For example, in the case of RWBF we also demand the following (omitted in figure 5):

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

   ```
   AG ((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 ])
   ```

   (The 'Weak until' connective, $\mathbf{W}$, is similar to $\mathbf{U}$ but $\varphi_1 \mathbf{W} \varphi_2$ does not require that φ_2 eventually become true if φ_1 is indefinitely true. One defines $\mathbf{A}[\varphi_1 \mathbf{W} \varphi_2]$ as $\neg \mathbf{E}[\neg\varphi_2 \mathbf{U} \neg(\varphi_1 \vee \varphi_2)]$.)

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

   ```
   AG ∀i ≠ j. ((ph[i].rbwf-number=j
                & ph[i].st=talking & ph[i].dialled=j)
               -> AF ph[i].rbwf-number=0)
   ```

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

```
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,busyt,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=busyt & 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=busyt : {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 5: The code for the Ring Back When Free feature

```
AG ∀i ≠ j. ((ph[i].rbwf-number=j & ph[i].st=talked
             & ph[j].dialled=i & ph[j].st=talking)
            -> AF ph[i].rbwf-number=0)
```

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

```
AG ∀i ≠ j. ((ph[i].rbwf-number=j
             & ph[i].st=talking & ph[i].dialled=j)
            -> AF ph[i].rbwf-use=0)
```

   RBWF is also deactivated when the target party calls.

```
AG ∀i ≠ j. ((ph[i].rbwf-number=j & ph[i].st=talked
             & ph[j].dialled=i & ph[j].st=talking)
            -> AF ph[i].rbwf-use=0)
```

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 3, and section C in the appendix.)

5.3 More features for the telephone 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 CFB to POTS+RBWF, 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.

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. 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.

With other features, however, RBWF is not always so well behaved. When we added Call Waiting to POTS+RBWF, we found that a number of requirements were violated. The first thing to notice was that CW did not respect the specifications introduced for RBWF. But on top of that, this combination of features also violated some of the requirements for CW. The violated requirements were concerned with the number of phones connected to another phone, for example: 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))
```

	CW	CFU	CFB	RBWF	RBWF[1]	TCS	OCS
CW	—	×	×	×	√	×	×
CFU	×	—	×	√	√	√	√
CFB	×	×	—	√	√	(√)	(√)
RBWF	×	√	√	—	√	√	√
RBWF[1]	√	√	√	√	—	√	√
TCS	×	√	(√)	√	√	—	√
OCS	×	√	(√)	√	√	√	—

Table 2: Interferences between features for the phone system

where `ph[1]` is the phone subscribing to CW and the flag `cw-msg` indicates whether the respective phone is on hold. The trace[3] 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

```
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 CW first and then RBWF, the system violates the RBWF requirements, 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 in case both features are activated.

Table 2 indicates interferences between features for the phone system. A plain tick denotes

[3]This trace has 17 states and is about 130 lines long!

Feature(s) \ Property (see appendix C)	Call from any phone to any other phone possible	Every call will end 'symmetrically'	Busy-tone or ringing-tone directly follow trying	Orig. line talking implies term. line talked to	A phone can have only one i/c call at a time	Dialled number cannot change w/out hanging up	CFU: phone will never ring	CFB: if engaged, i/c calls will term. at other line	CW: at most one party hears onhold msg.	CW: the 'active' party is never onhold	RBWF: call completion will be attempted	RBWF: stored number is reset on call completion	TCS: screened calls are never accepted	OCS: screened numbers cannot be contacted
POTS	√	√	√	√	√	√	—	—	—	—	—	—	—	—
CW	√	×	×	×	×	×	—	—	√	√	—	—	—	—
CFU	×	√	×	√	√	×	√	—	—	—	—	—	—	—
CFB	√	√	×	√	√	×	—	√	—	—	—	—	—	—
RBWF	√	√	√	√	√	√	—	—	—	—	√	√	—	—
TCS	×	√	√	√	√	√	—	—	—	—	—	—	√	—
OCS	×	√	√	√	√	√	—	—	—	—	—	—	—	√
CW + CFU	×	×	×	×	×	×	√	—	√	√	—	—	—	—
CFU + CW	×	×	×	×	×	×	√	—	√	×	—	—	—	—
CW + CFB	√	×	×	√	×	×	—	√	√	√	—	—	—	—
CFB + CW	√	×	×	√	×	×	—	×	√	×	—	—	—	—
CW + RBWF	√	×	×	×	×	×	—	—	×	√	√	√	—	—
RBWF + CW	√	×	×	×	×	×	—	—	√	√	×	√	—	—
CW * RBWF[1]	√	×	×	×	×	×	—	—	√	√	√	√	—	—
CW + TCS	×	×	×	×	×	×	—	—	√	√	—	—	√	—
TCS + CW	√	×	×	×	×	×	—	—	√	√	—	—	×	—
CW * OCS	×	×	×	×	×	×	—	—	×[4]	√	—	—	—	√
CFU + CFB	×	√	×	√	√	×	—	√	√	—	—	—	—	—
CFB + CFU	×	√	×	√	√	×	—	√	×	—	—	—	—	—
RBWF * CFU	×	√	×	√	√	×	√	—	—	—	√	√	—	—
TCS * CFU[2]	×	√	×	√	√	×	√	—	—	—	—	—	√	—
OCS * CFU[2]	×	√	×	√	√	×	√	—	—	—	—	—	—	√
RBWF * CFB	√	√	×	√	√	×	—	√	—	—	√	√	—	—
TCS * CFB[2]	×	√	×	√	√	×	—	×	—	—	—	—	√	—
OCS * CFB[2]	×	√	×	√	√	×	—	×	—	—	—	—	—	√
TCS * RBWF[3]	×	√	√	√	√	√	—	—	—	—	√	√	√	—
OCS * RBWF[3]	×	√	√	√	√	√	—	—	—	—	√	√	—	√

Table 3: Feature interactions for the telephone system

that there is no interference, i.e. that both features work correctly together and it does not matter in what order they are integrated. A tick in brackets indicates that the features commute (i.e. the integration order makes no difference) but that the features interfere with each other; and a cross means that they interfere and do not commute.

Table 3 summarises our 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; 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:
[1] Ring Back When Free subscribed to by a different phone.
[2] Call forwarding on Busy/Unconditional subscribed to by two phones.
[3] Call Screening subscribed to by two phones.
[4] This is clearly an artifact, generated by the use that Call Waiting makes of the variable `dialled`.

6 Conclusions

Our approach to the feature-interaction problem gives to features the status of first-class citizens; we could think of this as *feature orientation*. Concretely, this means that they are compact textual units in a specification or program, and that they are as independent as possible of the base system description. In this way, we develop a framework for *plug-and-play* features: features can be added, removed, re-ordered or re-designed in order to explore and resolve feature interactions.

Concerning the idea of a feature construct and feature integration in general, without specific reference to SMV, one may draw the following conclusions. Our case studies have shown features to be a useful design modularisation concept; we have described and verified a range of features. Our plug-and-play approach allowed us to explore feature interactions with an open mind, and we have witnessed surprising results. A negative point, however, visible even at this level of generality, is the dependence feature descriptions have on the base system.

From our experience of SMV, and our decision to define the feature construct for the SMV language, we draw mixed conclusions. On the positive side, the simple and intuitive syntax and semantics of SMV meant that our feature construct could also be simple and intuitive. However, SMV is not very expressive, and that also limits the expressiveness of the feature construct. Another problem with SMV was the state-space explosion problem: the time and space complexity of verifications is exponential in the number of variables, and this severely hampered our second case study (the phones). We were sometimes forced to be more abstract than we wanted to, in order to keep the verification tractable, and this meant we obtained some interactions which are really an artifact of our coding, while we missed some classical ones (namely those between Call Forwarding features and Call Screening Features). We nevertheless have obtained novel and significant data, and expect that our approach will benefit from advances in verification technology.

References

[1] G. C. Barney and S. M. dos Santos. *Elevator Analysis, Design and Control.* IEE Control Engineering Series 2. Peter Peregrinus Ltd., 1985.

[2] M. Berry. Proving properties of the lift system. Master's thesis, School of Computer Science, University of Birmingham, 1996.

[3] E.J. Cameron, N. Griffeth, Y.-J. Linand M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature interaction benchmark for in and beyond. In L. G. Bouma and Hugo Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 1–23, Amsterdam, The Netherlands, May 1994. IOS Press.

[4] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, number 803 in Lecture Notes in Computer Science, pages 124–175. Springer Verlag, 1993.

[5] M. R. Huth and M. D. Ryan. *Logic and its Practical Applications in Computer Science.* Book in preparation, 1998.

[6] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, 1993.

[7] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

Appendices

A A short introduction to SMV

Knowledge about SMV is required in order to understand our feature construct, our tool SFI, and the case studies presented in this paper. We apologize for leaving this exposition of SMV very sketchy and refer the interested reader to [4, 7] 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 (Computation Tree Logic).

It 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 the case that the formula is not satisfied, SMV also produces a trace showing circumstances in which the formula is false.

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.

```
 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 6: A system description for SMV

Figure 6 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 $(0, ready)$.

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.

Possible types for variables are boolean, enumerations (e.g. `state`), or finite ranges of integers. For declared variables (as opposed to `DEFINE`d 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 both ready or busy. (Since `request` is not determined at all by the description it, too, will assume values non-deterministically.) It is important to bear in mind that all assignments are evaluated in parallel (although there is also a mechanism for asynchronous (interleaving) composition of modules). A special kind of variable declaration is the the instantiation of a module, as in "`landingBut1 : button( lift.floor=1 & lift.door=open );`" (*cf.* figure 7 in appendix B). This is interpreted as a declaration of all local variables (including `DEFINE`d identifiers) of that module (`pressed` in this example), prefixed with the name of the newly declared variable, here `landingBut1.pressed`, together with the assignments or macro-definitions within that module. The formal parameters are replaced by the actual parameters as in a call-by-name language.

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). The propositional atoms for these formulas are boolean expressions over the variables of the system.

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

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \\ \mathbf{AX}\varphi \mid \mathbf{EX}\varphi \mid \mathbf{AG}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \mid \mathbf{E}[\varphi_1 \mathbf{U} \varphi_2].$$

where $p \in P$. (The other boolean operators ($\vee, \rightarrow, \leftrightarrow, \bot$) are defined in terms of $\wedge, \neg$ in the usual way.) In SMV, logical *or* is written as `|`, *and* as `&`, and *not* as `!`. Notice that CTL temporal operators come in pairs. The first of the pair is one of **A** and **E**. **A** means 'along all paths' (*inevitably*), and **E** means 'along at least one path' (*possibly*). The second one of the pair is **X**, **F**, **G**, or **U**, meaning 'neXt state', 'some Future state', 'all future states (Globally)', and Until respectively. Notice that **U** is binary. The pair of operators in $\mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$, for example, is **EU**. Further details of CTL are widely available in the papers by E. Clarke and others [4, 7], and also in the forthcoming introductory text [5].

B Further details for case study 1 (the lift system)

We describe the code to implement the basic lift system in SMV. The code is given in figures 7 to 9.

The module `main` (figure 7) declares five instances (one for each landing) of the module button (passing to each one as argument the conditions under which that button should cancel itself). It also declares one instance of `lift`, to which it passes a parameter whose value at any time is the next landing in the current direction of the lift which has requested the lift, and another parameter whose value at any time expresses whether there is a landing request.

The `lift` module (figure 9) 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 [1]: 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 9: 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

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

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 8), 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 pressed until the conditions to reset it arise inside the lift system.

B.1 Properties for basic lift system

These are the properties that we have verified for the base lift system, and some more that specify the behaviour of individual features. (We use the meta-notation introduced on page 6.)

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

Figure 8: The SMV code for the module `button` in the lift system.

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

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

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

   ```
   !AG (door=closed -> AF door=open)
   ```

4. The lift will not change its direction while there are calls in the direction it is travelling. One formula for upwards travel,

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

   ... and one formula for downwards travel, for $i > j$:

   ```
   AG ∀i > j. (floor=i & liftButj.pressed & direction=down
               -> A[direction=up U floor=j])
   ```

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

   ```
   EF (floor=i & door=closed & idle)
   AG (lift.floor=i & lift.idle & lift.door=closed
       -> EG (lift.floor=i & lift.door=closed))
   ```

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

   ```
   ∀i ∈ {2,3,4}. !AG ((floor=i & !liftButi.pressed & direction=up)
                      -> door=closed)
   ∀i ∈ {2,3,4}. !AG ((floor=i & !liftButi.pressed & direction=down)
                      -> door=closed)
   ```

   ... and at floors 1 and 5 regardless of direction:

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

```
!AG ((floor=1 & !liftBut1.pressed) -> door=closed)
!AG ((floor=5 & !liftBut5.pressed) -> door=closed)
```

B.2 Properties for the featured lift system

In addition to these generic properties, we check some requirements for each feature.

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

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

   The lift will honour requests from within the lift as long as it is not empty.

   ```
   AG ∀i. (lift.liftButi.pressed & !lift.empty)
          -> AF ((lift.floor=i & lift.door=open) | lift.empty)
   ```

2. Overloaded:
   The doors of the lift cannot be closed when the lift is overloaded.

   ```
   !EF (overload & door=closed)
   ```

   The lift will not move while it is overloaded.

   ```
   AG (lift.floor=i & lift.overload
       -> A[ lift.floor=i W !lift.overload ])
   ```

3. Parking:
   The lift will not remain idle indefinitely at any floor other than floor 1.

   ```
   AG ∀i ≠ 1. !EG(floor=i & door=closed)
   ```

4. Lift-$\frac{2}{3}$-full:
   Car calls have precedence when the lift is $\frac{2}{3}$ full (indicated by the flag `tt-full`).

   ```
   AG ∀i ≠ j. ((lift.tt-full &
                lift.liftButi.pressed & !lift.liftButj.pressed)
               -> A [!(lift.floor=j & lift.door=open)
                     U ((lift.floor=i & door=open)
                       | !lift.cp | lift.liftButj.pressed)])
   ```

5. Executive Floor:
   The lift will answer requests from the executive floor (`lift.ef`).

   ```
   AG (lift.ef=i
       -> A[ (landingButi.pressed -> AF(lift.floor=i))
           W !lift.ef=i ])
   ```

   Here again we use the 'weak until' connective, to allow for executions where `lift.ef` never changes. (*Cf.* page 11)

C Further details for case study 2 (the telephone system)

The code for the phones is given in figures 10 and 11.

C.1 Properties for 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 6.)

1. Any phone may call any other phone.

   ```
   AG ∀i ≠ j. (EF (ph[i].st=talking & ph[i].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 (`st=idle`) and the other party still off-hook (`st=ended`). (This holds only with "weak" fairness, which ensures that a phone cannot remain in the same state indefinitely.)

   ```
   AG ((ph[i].dialled=j & ph[i].st=talking)
           -> AF ((ph[i].st=idle & ph[j].st=ended) |
                  (ph[j].st=idle & ph[i].st=ended)))
   ```

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

   ```
   AG (ph[i].st=trying -> AX (ph[i].st=ringingt | ph[i].st=busyt))
   ```

4. 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.

   ```
   AG ((ph[i].st=trying & ph[i].dialled=j)
           -> AX (ph[i].st=ringingt -> ph[j].st=ringing))
   ```

5. A list of SPECs stating that if a phone is talking, the dialled phone must be talked to.

   ```
   AG (ph[i].st=talking & ph[i].dialled=j -> ph[j].st=talked)
   ```

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

   ```
   EF ph[i].st=talked
   AG (ph[i].st=talked
          <-> ∃j.(ph[j].st=talking & ph[j].dialled=i))
   ```

7. Phone i can be ringing; and if it is ringing, there has to be another phone that has dialled it and is getting the ringing-tone.

   ```
   EF ph[i].st=ringing
   AG (ph[i].st=ringing
          <-> ∃j.(ph[j].st=ringingt & ph[j].dialled=i))
   ```

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

```
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
                     st=idle                : 0;
                     st=dialt & dialled = 0 : {1,2,3,4};
                     1                      : dialled;
                   esac;

  init(st) := idle;
  next(st) :=
    case
      st=idle :
        case
          p[2].st=trying & p[2].dialled=1 & next(p[2].st)=ringingt : ringing;
          p[3].st=trying & p[3].dialled=1 & next(p[3].st)=ringingt : ringing;
          p[4].st=trying & p[4].dialled=1 & next(p[4].st)=ringingt : ringing;
          1                                                        : {idle, dialt};
        esac;

      st=ringing :
        case
          p[2].st=ringingt & p[2].dialled=1 & next(p[2].st)=idle : idle;
          p[3].st=ringingt & p[3].dialled=1 & next(p[3].st)=idle : idle;
          p[4].st=ringingt & p[4].dialled=1 & next(p[4].st)=idle : idle;
          1                                                      : {ringing, talked};
        esac;

      st=dialt : {idle, dialt, trying};

      st=busyt : {idle, busyt};

      st=trying :
        case
          dialled=2 & p[2].st=idle & next(p[2].st)=ringing
          & ((p[3].st=trying & p[3].dialled=2)->next(p[3].st)=busyt)
          & ((p[4].st=trying & p[4].dialled=2)->next(p[4].st)=busyt) :ringingt;
          dialled=3 & p[3].st=idle & next(p[3].st)=ringing
          & ((p[2].st=trying & p[2].dialled=3)->next(p[2].st)=busyt)
          & ((p[4].st=trying & p[4].dialled=3)->next(p[4].st)=busyt) :ringingt;
          dialled=4 & p[4].st=idle & next(p[4].st)=ringing
          & ((p[2].st=trying & p[2].dialled=4)->next(p[2].st)=busyt)
          & ((p[3].st=trying & p[3].dialled=4)->next(p[3].st)=busyt) :ringingt;
          1                                                          :busyt;
        esac;

      st=ringingt :
        case
          dialled=2 & next(p[2].st)=talked : talking;
          dialled=3 & next(p[3].st)=talked : talking;
          dialled=4 & next(p[4].st)=talked : talking;
          1                                : {ringingt, idle};
        esac;
```

Figure 10: The SMV code for the phone system. (1/2)

```
      st=talked :
        case
          p[2].st=talking & p[2].dialled=1 & next(p[2].st)=idle : ended;
          p[3].st=talking & p[3].dialled=1 & next(p[3].st)=idle : ended;
          p[4].st=talking & p[4].dialled=1 & next(p[4].st)=idle : ended;
          1                                                    : {idle, talked};
        esac;

      st=talking :
        case
          dialled=2 & p[2].st=talked & next(p[2].st)=idle : ended;
          dialled=3 & p[3].st=talked & next(p[3].st)=idle : ended;
          dialled=4 & p[4].st=talked & next(p[4].st)=idle : ended;
          1                                               : {idle, talking};
        esac;

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

DEFINE
  party :=
    case
      st=talking                      : dialled;
      p[B].st=talking & p[B].dialled=X : B;
      p[C].st=talking & p[C].dialled=X : C;
      p[D].st=talking & p[D].dialled=X : D;
      1                               : 0;
    esac;

-- Fairness constraints to ensure that a phone does not remain in a state
-- indefinitely.  A phone may still alternate between,  say,  idle <-> 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 11: The SMV code for the phone system. (2/2)

```
AG ∀i ≠ j. !(ph[i].st=talking & ph[i].dialled=k &
             ph[j].st=talking & ph[j].dialled=k )
```

9. The dialled number cannot change without replacing the hand-set. (This only holds with "weak" fairness, otherwise one has to use the 'weak until' connective, *cf.* page 11.)

```
AG ((ph[i].dialled=j & ph[i].st=trying)
  -> (A[ ph[i].dialled=j U ph[i].st=idle]))
```

C.2 Properties for the featured phone system

In addition to these generic properties, we give some of the requirements we verified for each feature.

1. Call Forwarding Unconditional
   If a forwarding number is given, the phone will never ring. (The forwarding number is chosen at random but does not change after that.)

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

2. 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 ∀i ≠ j ≠ k. ((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=ringing)))
```

3. Call Waiting
   If there are two calls, exactly one party will hear the 'onhold'-message. (In other words, at most one party will hear the 'onhold'-message.)

```
AG ∀i ≠ j ≠ k. (ph[i].st=talking & ph[i].dialled=k &
                ph[j].st=talking & ph[j].dialled=k
                -> (ph[i].cw-msg <-> !ph[j].cw-msg))
```

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

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

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

```
AG ((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 ])
```

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

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

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

8. Originating Call Screening
   Calls to numbers on the screening list (array `ocs`) never succeed.

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