In *Advances in Computer Science*, 1999, to appear.

SFI: a Feature Integration Tool*

Malte Plath and Mark Ryan

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.

To support this way of building a system from a basic system by successively adding features, we have developed the tool SFI ('SMV feature integrator') that automates the integration of features into a formal description of the 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. We handle the potential inconsistency between a feature and the base system by allowing features to override existing behaviour in a tightly controlled way.

As a first case study, we have analysed the *lift system* and its features. The base lift system consists of a lift which responds to requests made by pressing buttons by moving up and down between floors and opening its doors. It works according to an algorithm known as Single Button Collective Control (SBCC) (Barney and dos Santos 1985). We present another case study, based on the telephone system, in a related paper (Plath and Ryan 1998).

We have used our method and tool to extend the basic lift system with several features that are found in more sophisticated lift systems, such as: parking on the ground floor in anticipation of demand; detecting overloads; executive floor priorities; etc. Our method can be used to investigate the circumstances such features conflict, or do not work as intended.

The remainder of the paper is structured as follows. The next section describes our approach to features and feature-integration. In section 3, we briefly introduce SMV; the reader is referred to other papers for a fuller explanation. Section 4 contains details of our tool, called SFI, and section 5 describes a case study in detail. We conclude in section 6.

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

2 Features and feature-integration

A feature is a small increment in functionality to a base system. For example, the lift's ability to detect that it is overloaded, and to behave differently in those circumstances, is a feature for the basic lift system. Software systems are often upgraded by adding features.

The 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. Feature descriptions try to avoid making assumptions about the architecture of the base system in question.

In our approach, features are defined as self-contained textual units, in a manner which makes them easy to understand in isolation. A feature description can be seen as a prescription for changing and extending the basic system. The process of integrating a feature into an existing system is automatic, rather like applying a patch. Thus, they are easy to add to a system, or to remove, or to re-specify. A feature usually overrides some old behaviour of the system; our approach provides a clear mechanism for this. A feature may be thought of as a particular case of superimposition (Katz 1993).

Features are intended to be used for structuring a system description like (optional) modules, only that they lack their compositional character ("modularity"). The specifier/developer of a feature ideally should refer only to the code for the base system in the development of the feature. (Unless, of course, the feature builds on an already feature-extended version of that system.)

The tool SFI is designed for use with the SMV description language (see (Clarke et al. 1993) or (McMillan 1993)). That is, SFI takes a base system description written in SMV and some features written in the extension of SMV that we define here, and returns a description in SMV representing the integration of the features into the base system. Our approach is quite general, however, and need not be tied to any particular system description language. A similar tool could be developed for other description languages.

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

- It is designed and optimised for concurrent, reactive systems, such as the telephone system and the lift system. The feature interaction problem, which we wish to investigate with our approach, arose originally in such systems. (*Cf.* (Griffeth 1992), (Bouma and Velthuijsen 1994), (Cheng and Ohta 1995), (Dini et al. 1997) and (Kimbler and Bouma 1998))
- The SMV tool (McMillan 1993) can check temporal properties of systems described using the SMV language. This enables rapid development of rigorous and accurate examples. We use the SMV tool both before and after feature integration with SFI.

Feature integration is the process by which a new system is constructed from an old system and a feature; this is carried out automatically by our tool SFI. Automating this process means that we can study questions such as: is a given

feature compatible with this system (i.e. is the integration possible)? Does it matter in what order we integrate two features into a system? (The answer is usually yes; a negative answer would show that the two features are in some sense independent with respect to the system.) Does integrating feature A prevent or interfere with the later integration of feature B? Does the integration of B break the functionality provided by a previously integrated feature A? These questions are made precise below in section 5.3.

3 A very short introduction to SMV

Before we explain how feature integration with SFI works, we should say a few words about SMV[2]. We apologise for leaving this very sketchy and refer the interested reader to (Clarke et al. 1993) and (McMillan 1993) for a more detailed account. SMV is a CTL[3] model checker for (unlabelled) finite automata. It takes the description of an automaton in the SMV language and some properties in the form of CTL formulae. The following is one of the examples distributed with the SMV system. (The line numbers are not part of the code.)

```
 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)
```

Fig. 1. A system description for SMV

This piece of code defines a non-deterministic 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`) and assignments (`ASSIGN`). 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

[2] Symbolic Model Verifier
[3] Computational Tree Logic

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` equals 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. However, modules can be composed both asynchronously and synchronously.

After defining a system in the SMV language, we formulate the properties to be verified in the temporal logic CTL (`SPEC`, e.g. line 11). The propositional atoms for these formulae 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 the machine readable form `&`, `|` and `!` are used instead of $\wedge, \vee$ and $\neg$, respectively.)

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** . Details of CTL are widely available in the papers by E. Clarke and others (for example (Clarke et al. 1993) and (McMillan 1993)), and also in the forthcoming introductory text (Huth and Ryan 1998).

A useful derived connective is **AW** , which uses the 'weak until' connective **W**, which is similar to **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 $\neg\mathbf{E}[\neg\varphi_2 \mathbf{U} \neg(\varphi_1 \vee \varphi_2)]$.

4 SFI – a feature integration tool

In this section, we present an extension of the SMV syntax for describing features. We describe SFI, our tool for compiling programs written in the extended SMV into pure SMV, thus giving semantics to the feature construct. We illustrate its semantics by developing the examples of features for the lift system.

4.1 Syntax and semantics of the feature construct

A formal specification of the syntax is given in figure 2. There are three main sections of the feature construct, introduced by the keywords `REQUIRE`, `INTRODUCE` and `CHANGE`.

The `REQUIRE` and `INTRODUCE` sections deal with the vocabulary used by the feature. 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

```
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 ] }*
]
```

where:

impose-clause stands for
    IMPOSE *assignments*

treat-clause stands for
    TREAT var_1 = $expr_1$ [, ... var_n = $expr_n$]

[ ] stands for 'optional'

[| |] stands for 'one of'

{ }* stands for 'several'

Fig. 2. The syntax of the feature construct

modules and variables in modules may be specified there. The INTRODUCE section states what new modules or new variables within old modules are introduced by integrating the feature into a program. SMV DEFINE and ASSIGN clauses may also be given, and CTL formulas in SPEC clauses may be given. These are textually added to the program at integrate-time. All old modules and variables that are used in the INTRODUCE section should be REQUIREd, and their absence will lead to an error.

The CHANGE section specifies what the feature actually does, by introducing a mask or wrap on top of the base program's code. It allows the programmer to describe certain kinds of changes to the program by affecting the way variables are read and written. The CHANGE section gives a number of TREAT or IMPOSE clauses, which may be guarded by a condition. These clauses allow the feature to interfere with variables maintained by the base system. The IMPOSE clause allows a feature to overwrite the current value of a variable with another value. The TREAT clause enables a feature to mask the actual value of a variable with a different value. It does not change the actual value, but arranges matters so that when the variable is accessed (and the guarding condition is true), the actual

value is hidden and a different value is returned. The precise meaning of `TREAT` and `IMPOSE` is given below.

4.2 What SFI does

SFI takes a base system description written in SMV and a feature written using the syntax of figure 2, and returns a new SMV description which represents the integration of the feature into the base. It checks the presence of the entities specified in the `REQUIRE` section, and inserts the code given in the `INTRODUCE` section in the appropriate places. It then alters the code of the base system in the way prescribed by the `CHANGE` section of the feature, as follows:

- For `CHANGE`s of the form
  `IF` *cond* `THEN TREAT` x `=` *expr*
  Replace 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 e. 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*, replace *oldexpr* by

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

  Whereas `TREAT` just deals with expressions reading the value of x, i.e. occurrences on the right-hand-side of an assignment to another variable, `IMPOSE` deals with assignments to the variable x. It has the effect that, while *cond* is true, x is only 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.
- 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 are replaced directly by the new expression (*expr*).

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

- The elements 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 introduced in 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.

We have implemented the integrator in C. It accepts as input SMV text and feature descriptions; it performs the necessary checks and substitutions and then writes out a new SMV source file.

5 The lift system Case study and experimental results

As a first case study, we have analysed the *lift system* and its features. We first describe the base lift system; then we look at its features.

5.1 The base system, SBCC

The base lift system consists of a single lift. It accepts requests made by users pressing buttons on the floor landings and from inside the lift. It moves up and down between floors and opens and closes its doors in response to those requests. It works according to an algorithm known as Single Button Collective Control (SBCC) (Barney and dos Santos 1985). In essence, the lift travels in its current direction to the next floor at which there is a request, and opens its doors; then it closes its doors and continues in the current direction, until there are no more requests pending in that direction. Then it reverses its direction of travel, and proceeds as before. 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.

The SMV code for the base lift system is shown in figures 3 to 5, adapted from (Berry 1996). The module `main` (figure 3) 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 5) declares the variables `floor`, `door` and `direction` as well as a further 5 buttons, this time those inside the lift.

By inspecting the `button` module (figure 4), one finds that it sets its variable `pressed` 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.

```
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);
```

Fig. 3. The SMV code for the module `main` in the lift system.

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

Fig. 4. The SMV code for the module `button` in 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;
```

Fig. 5. The SMV code for the module `lift` in the lift system.

Properties for basic lift system. Before any features are added, we may use SMV to check the following basic properties of the lift system.[4] These properties are also checked against the featured systems, to test whether the features violate them. (See also table 1.)

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

3. The lift will not change its direction while there are calls in the direction it is travelling. The first part of the formula is for upwards travel, and the second part for downwards travel:

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

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

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

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:

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

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

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

[4] 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. In these examples, any free variables are universally quantified. For example, if we expand the first specification below 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))
```

5.2 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, in order to optimise performance. This is because the next request is anticipated 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. Code for the parking feature is given in figure 6. In our example, it parks on the ground floor (i.e. floor number 1) by treating the situation as if there was a call to that floor whenever there are no pending requests.

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. The code for this feature is given in figure 7. It introduces a new boolean variable `overload` (signifying that the lift is overloaded) and imposes that the doors are always open when `overload` is true.

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.

Our method provides a framework to plug these different features into the lift system, and by examining the result, to witness interactions and interferences between features. 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 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 (1–7) which apply to any lift system (featured or not), and have been described above. We can see which properties are broken by the addition of various features. To the right of the double line are the properties (8–14) which are designed to test the integration of specific features.

```
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
```

Fig. 6. The code for the Parking feature

Whenever a property is violated by a (featured) system, SMV prints out a trace showing how the violation can occur. In most cases such traces illustrate possible "pathological" behaviours quite succinctly.

In the following we list the properties which were checked for the featured systems. Each property corresponds to a requirement of a feature and hence serves to test the correct operation of that feature.

Properties for the featured lift system.

8. 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`
   `    -> landingBut`i`.pressed)`
9. Empty:
   The lift will honour requests from within the lift as long as it is not empty.
   `AG ∀`i`. (lift.liftBut`i`.pressed & !lift.empty)`
   `       -> AF ((lift.floor=`i` & lift.door=open) | lift.empty)`

```
FEATURE overloaded
REQUIRE
  MODULE lift
  VAR door : {open, closed};
      floor : 1..4;

INTRODUCE
  MODULE lift
  VAR overload : boolean;
  ASSIGN overload := case
                       door = closed : overload;
                       1             : {0,1};
                     esac;
  -- the lift will not move while overloaded
  SPEC  AG (floor=1 & overload -> A [floor=1 U !overload])
  -- the doors cannot be closed if the lift is overloaded
  SPEC  !EF (overload & door=closed)

CHANGE
  MODULE lift
  IF overload THEN IMPOSE door:=open;
```

Fig. 7. The code for the Overloaded feature

		Property													
Feature 1	Feature 2	1	2	3	4	5	6	7	8	9	10	11	12	13	14
none	none	√	√	√	√	√	√	√	—	—	—	—	—	—	—
Empty	none	√	×	×	√	√	√	√	√	√	—	—	—	—	—
Overloaded	none	×	×	×	√	√	√	√	—	—	√	√	—	—	—
Parking	none	√	√	√	√	×	√	√	—	—	—	—	√	—	—
Lift-$\frac{2}{3}$-full	none	×	√	√	√	√	√	√	—	—	—	—	—	√	—
Exec. Floor	none	×	×	√	√	√	√	√	—	—	—	—	—	—	√
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	×	×	√	√	×	√	√	—	—	—	—	—	×	×

Explanation of symbols: √ property holds
× property violated
— property not applicable

Properties are numbered as in sections 5.1 and 5.2

Table 1. Feature interactions for the lift system

10. Overloaded:
    The doors of the lift cannot be closed when the lift is overloaded.
    ```
    !EF (overload & door=closed)
    ```
11. Overloaded:
    The lift will not move while it is overloaded.
    ```
    AG (lift.floor=i & lift.overload
        -> A[ lift.floor=i W !lift.overload ])
    ```
12. Parking:
    The lift will not remain idle indefinitely at any floor other than floor 1.
    ```
    AG ∀i ≠ 1. !EG(floor=i & door=closed)
    ```
13. Lift-$\frac{2}{3}$-full:
    Lift 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)])
    ```
14. 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 ])
    ```

5.3 Feature interaction

A significant motivation for the feature construct introduced in this paper is the concept of feature interaction. When several features are integrated on top of a base system, they may interfere with each other, or interact in ways which are hard to predict. This problem has been dubbed the *feature interaction problem* in the literature on telecommunications. A series of workshops is dedicated to feature interaction: (Griffeth 1992), (Bouma and Velthuijsen 1994), (Cheng and Ohta 1995), (Dini et al. 1997) and (Kimbler and Bouma 1998).

We write $S+F$ for the result of integrating the feature F into the base system S; and we write $S \models \varphi$ to mean that the system S satisfies the property φ. Our method and tool can detect the following kinds of feature interaction.

Non-commutativity of features. If the order of integration matters, i.e. if $(S + F_1) + F_2$ and $(S + F_2) + F_1$ satisfy different specifications, we say that F_1 and F_2 are not commutative with respect to S.
In the case of the lift system, every pair of features is commutative except Lift-$\frac{2}{3}$-full and Executive-floor. As shown in table 1, integrating Executive-floor first and then Lift-$\frac{2}{3}$-full satisfies the specification that lift calls take precedence when the lift is $\frac{2}{3}$-full; but integrating them the other way around does not. Intuitively, this is because, in the second case, an executive may call the lift when it is $\frac{2}{3}$-full, and that request will satisfied. Indeed, the trace output by SMV for this property failure shows such a scenario.

Violation of a property introduced by another feature. Let φ_1 and φ_2 be properties introduced by features F_1 and F_2, respectively. Independently of whether F_1 and F_2 commute or not, we may find that

- $S + F_1 \models \varphi_1$, but $(S + F_1) + F_2 \not\models \varphi_1$. In this case, F_2 interferes with a previously applied feature, F_1, preventing it from working correctly.
- $S + F_2 \models \varphi_2$, but $(S + F_1) + F_2 \not\models \varphi_2$. In this case, F_1 prevents the successful integration of a later feature F_2.

The non-commuting pair of features Lift-$\frac{2}{3}$-full and Executive-floor are an example of the first kind of these interactions. The pairs

(Overloaded, Empty),
(Lift-$\frac{2}{3}$-full, Empty),
(Executive-floor, Empty),
(Lift-$\frac{2}{3}$-full, Overloaded),
(Executive-floor, Overloaded),

are each commutative, and are examples of both these types of interaction.

Joint violation of base property. This kind of interference occurs if $(S + F_1) + F_2$ violates specifications of the base system that hold for both $S + F_1$ and $S + F_2$. The features Lift-$\frac{2}{3}$-full and ExecutiveFloor are an example of such an interaction; when integrated in that order, the resulting system violates the property that the lift can remain idle at any floor.

6 Conclusions and future work

We support the user's natural tendency to think of a sophisticated software system as a base system together with a collection of features, by providing a feature construct for the SMV description language. We have implemented a tool, called SFI, which compiles the feature construct into pure SMV. The lift system provides a case study, for which we specify five features and explore their behaviours and interferences using SFI. Our method detects a variety of interactions.

The construct for defining features has proved to be both intuitive and powerful; witness the variety of features we have defined. The `TREAT` and `IMPOSE` clauses are able to express the impact the feature has on the underlying code in a natural way, and the feature specifier is not too tied to details of the underlying base system. This is in part due to the simple and intuitive syntax and semantics of SMV; we anticipate that defining a feature construct for more sophisticated languages will be more complicated.

References

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

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

Bouma, L. G. and Velthuijsen, H., editors (1994). *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands. IOS Press.

Cheng, K. E. and Ohta, T., editors (1995). *Feature Interactions in Telecommunications III*, Tokyo, Japan. IOS Press.

Clarke, E., Grumberg, O., and Long, D. (1993). 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.

Dini, P. et al., editors (1997). *Feature Interactions in Telecommunications and Distributed Systems IV*, Montreal, Canada. IOS Press.

Griffeth, N., editor (1992). *1st International Workshop on Feature Interactions in Telecommunications Software Systems*, St. Petersburg, Florida, USA.

Huth, M. R. and Ryan, M. D. (1998). *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press. Book in preparation.

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

Kimbler, K. and Bouma, L. G., editors (1998). *Feature Interactions in Telecommunications and Software Systems V*, Lund, Sweden. IOS Press.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

Plath, M. C. and Ryan, M. D. (1998). Plug-and-play features. In (Kimbler and Bouma 1998), pages 150–164.