Feature integration using a feature construct [1]

Malte Plath and Mark Ryan

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

Abstract

A *feature* is a unit of functionality that may be added to (or omitted from) a system. Examples of features are plug-ins for software packages or additional services offered by telecommunications providers. Many features override the default behaviour of the system, which may lead to unforeseen behaviour of the system; this is known as *feature interaction.*

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 is given for integrating a feature into a system description. It allows features to override existing behaviour of the system being developed;
- A prototype tool has been developed for performing the integration;
- Interactions between features may 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: a lift system and a telephone system.

Key words: features, feature interaction, model checking

[1] Financial support from the EU through Esprit working groups ASPIRE (22704) and FIREworks (23531), and from British Telecommunications PLC and the Nuffield Foundation in the UK is gratefully acknowledged.

1 Introduction

The concept of *feature* has emerged in the telecommunications industry 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; serial interface; ability to print double-sided; and others. Thinking 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.

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 ([11], [3], [4], [8] and [17]).

Examples of feature interactions in telecommunications systems are:

- "Call Forward on Busy" and "Voice Mail on Busy": both these features try to take control of a second (incoming) call to the subscriber. This is inconsistent, so one cannot allow both features to be active on the same phone.
- The "Ring Back When Free" (RBWF) attempts to set up a call for the subscriber to a callee whose line is engaged as soon as the line becomes free. The interaction of RBWF and "Call Forward Unconditional" (CFU) leads to consistent, but potentially undesirable behaviour:
  x requests RBWF from **y**, then CFU to **z**.
  z will be notified when **y** becomes available.
  However, it was **x** who requested the notification.

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 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 packages are `german.sty` and `amstex.sty` (when 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 if the features we allow are reasonably powerful. 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[2] into the telephone system, 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.

But feature interaction is not always a bad thing — take the second example above: **x** may be forwarding her calls to her mobile phone, in which case the indicated behaviour is desirable. (In this case **x** and **z** refer to the same person.) In cases when it is clear that the result of a feature interaction is undesirable, or detrimental to the operation of the whole system, we often use the term feature *interference* to stress this fact.

Since there is no way of avoiding feature interaction short of rigidly restricting the set of potential features, it is desirable 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 [6,19]. 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 section 2.

We extend the SMV language with a new construct for describing features.

[2] For the situation when a call arrives at an engaged phone: the new caller will *not* get the busy tone, and the call can be completed.

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 the integration process in the remainder of the paper, along with two substantial case studies of feature integration. This paper extends and completes two previous papers [21,22].

The structure of this paper is as follows: in the following section we give a short introduction to the SMV language. In section 3 we describe the ideas behind our approach. This is followed by an explanation of the feature construct for SMV in section 4. Sections 5 and 6 are devoted to our two case studies, the lift system and the telephone system, respectively. The files for these case studies can be found at `http://www.cs.bham.ac.uk/~mdr/features/SFI-demo/`. We conclude our paper by summing up our experiences with this approach in section 7.

2 A short introduction to SMV

Knowledge about SMV[3] is required in order to understand our feature construct, our tool SFI, and the case studies presented in this paper. We apologise for leaving this exposition of SMV very sketchy and refer the interested reader to [6,19] 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 symbols we write $S \models \phi$ if the system S satisfies the formula ϕ, and $S \not\models \phi$ if ϕ does not hold for S. 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,

[3] Until 1998 there was just one SMV, but now there are three. CMU SMV [19] is the original one, developed by Ken McMillan, and is the one we use in this paper. NuSMV is a re-implementation being developed in Trento [5], and is aimed at being customisable and extensible. Cadence SMV is an entirely new model checker focussed on compositional systems. It is also developed by Ken McMillan, and its description language resembles but much extends the original SMV [18].

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)
```

Fig. 1. A system description for SMV

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

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 ($\{0,1\}$), enumerations (e.g. `state`), finite ranges of integers, or arrays of these types. 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 either *ready* or *busy*. (Since `request` is not determined at all by the description, it too will assume values non-deterministically.) Note that *case statements* are evaluated top to bottom, so the result is the expression from first branch whose condition evaluates to true. 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 instantiation of a module, as in "`landingBut1 : button(lift.floor=1 & lift.door=open);`" (*cf.* figure 3 in section 5). This is interpreted as a declaration of all local variables (including `DEFINE`d identifiers) of that module, prefixed with the name of the newly declared variable, together with the assignments or macro-definitions within that module. In this example, the module `button` has a local variable `pressed`, so the declaration above implicitly declares `landingBut1.pressed`. The formal parameters are replaced by the actual parameters as in a call-by-name language.

It is possible to assert fairness constraints on the model (*cf.* figure 9, page 24). In the presence of such fairness constraints, only executions are considered along which these constraints are true infinitely often.

After defining a system in the SMV language, we formulate the properties to be verified in the temporal logic CTL (marked by the keyword `SPEC`, e.g. line 11). The propositional atoms for these formulas are the boolean variables and the equations over the variables and constants of the system.

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

$$\begin{aligned}\phi ::= & p \mid \top \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \\ & \mathbf{AX}\phi \mid \mathbf{EX}\phi \mid \mathbf{AG}\phi \mid \mathbf{EG}\phi \mid \mathbf{AF}\phi \mid \mathbf{EF}\phi \mid \mathbf{A}[\phi_1\mathbf{U}\phi_2] \mid \mathbf{E}[\phi_1\mathbf{U}\phi_2].\end{aligned}$$

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 `!`; truth ($\top$) is represented by `1` and falsity ($\bot$) by `0`.

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 $\mathbf{E}[\phi_1\mathbf{U}\phi_2]$, for example, is **EU**. Further details of CTL are widely available in the papers by E. Clarke and others [6,19], and also in the forthcoming introductory text [14].

Two useful derived connectives are **AW** and **EW**, which use the 'weak until' connective **W**, which is similar to **U**, but $\phi_1\mathbf{W}\phi_2$ does not require that ϕ_2 eventually becomes true if ϕ_1 is indefinitely true. One defines $\mathbf{A}[\phi_1\mathbf{W}\phi_2]$ as $\neg\mathbf{E}[\neg\phi_2\mathbf{U}\neg(\phi_1 \vee \phi_2)]$, and $\mathbf{E}[\phi_1\mathbf{W}\phi_2]$ as $\mathbf{E}[\phi_1\mathbf{U}\phi_2] \vee \mathbf{EG}\phi_1$.

3 Features and feature integration

In this section we describe how an existing system can be extended and altered to provide new functionality. The new functionality will be given in the form of features that are integrated into the system. In this sense every feature can be seen as a packet of functionality. We can also see it as a transformation of the old system to a new one which offers different functionality. Formally we distinguish between these aspects by calling the transformation *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, i.e. interactions which are inherent in the specification and quite independent of the implementation (e.g. inconsistencies), this is not necessarily a shortcoming, for at a high level of abstraction the logical interactions may 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 [6,19]. 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. Elsewhere [23] we have applied it to the model checker SPIN [13] and its language Promela. We are also developing a feature construct for CSP [12], using the model checker FDR2 [10].

SMV is well-suited to this approach for the following reasons:

- The SMV language is designed and optimised for concurrent, reactive systems, such as the telephone system.
- The SMV language is expressive yet compact. Its compactness means that the feature construct is compact too, and feature integration is relatively straight forward.
- 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* [16]. 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 [16] is suited to imperative languages, and therefore cannot be used directly for SMV. In imperative languages data and control flow are explicit, and the superimposition construct works by modifying them. For a declarative language like SMV data and control flow are implicit. An SMV program essentially is just a set of assignments to state variables. Hence, state variables and assignments are the entities which a superimposition or feature construct for SMV has to be based on. In the following section we will show how this is done.

4 The feature construct for SMV

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, 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 2. 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.

```
FEATURE feature-name
[ REQUIRE
    { MODULE module-name [ (parameter-list) ]
        VAR variable-declarations }*
]
[ INTRODUCE
    { MODULE module-name
        [ VAR variable-declarations ]
        [ ASSIGN assignments ]
        [ DEFINE definitions ]
        [ { SPEC formula }* ] }*
]
[ CHANGE
    { MODULE module-name
        [ IF condition THEN ]
              [ IMPOSE assignments
              | TREAT $var_1$ = $expr_1$ [, ...$var_n$ = $expr_n$ ]
              ] }*
]
END
```

where: [ ] stands for 'optional'
[| |] stands for 'one of'
{ }* stands for 'several'

Fig. 2. The syntax of the feature construct

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 how the feature changes the behaviour of the system wrt. the original state variables. It gives a number of `TREAT` or `IMPOSE` clauses, which may be guarded by a condition. This is where the behaviour of the original system is altered.

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

- It checks that the `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 because x may occur in conditions in case statements, and SMV requires such conditions 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.

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

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; a detailed account can be found in [20].

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

In the next two sections, we explore feature integration in the context of our case studies. This will illustrate the effect of integrating features in different orders.

Detecting feature interaction. We view a feature as comprising two components: the feature implementation and the feature requirements. In the following we write (F, ϕ) for a feature. In practice it is usually more useful to state the requirements as several formulas. The formula ϕ then stands for a specific property one would like to verify of the system. When we integrate a feature (F, ϕ) into a base system S, to yield a new system $S + F$, we want to test the following:

- $S + F \models \phi$: Feature F has been successfully integrated.
- $(S + F_1) + F_2 \models \phi_2$: Feature F_2 can be integrated into the extended system $S + F_1$.
- $(S + F_1) + F_2 \models \phi_1$: Feature F_2 does not violate the requirements of F_1.

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

- Type I: $(S + F_1) + F_2 \not\models \phi_2$:
  Earlier feature breaks later one.
- Type II: $(S + F_1) + F_2 \not\models \phi_1$:
  Later feature breaks earlier one.
- Type III: $S, S + F_1, S + F_2 \models \psi$ but $(S + F_1) + F_2 \not\models \psi$:
  (where ψ is a property of the base system.)
  Features combine to break system.
- Type IV: $\exists \phi.(S + F_1) + F_2 \models \phi$ but $(S + F_2) + F_1 \not\models \phi$:
  (where ϕ is a property of S, F_1 or F_2)
  Features do not commute.

Note that these types of interactions do not represent a disjoint classification; two features may exhibit several types of interaction. Obviously, for commuting features, a Type I interaction for integration of F_1 and then F_2 corresponds to a Type II interaction for the reverse order of integration, and vice versa.

We will come back to this classification in the analysis of our case studies.

5 Case study 1: the lift system

5.1 The basic lift system

As a first case study, we have analysed a *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 figures 3 to 5. It consists of about 120 lines of SMV code.

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 two parameters:[4]

- the next landing – in the current direction of travel – at which there was a request for the lift,
- and whether there is a landing request.

The `lift` module (figure 4) 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 4: if the lift is idle, it maintains the same direction as it had before, but if it is at the top or bottom of its shaft it changes direction to down and up respectively; otherwise, as stated, it reverses direction if there are no calls remaining to be served in the current direction. The final '`1:direction`' means that if none of the preceding conditions are true, then the value returned by the case statement is simply the old value of direction. Notice that the SBCC algorithm stipulates only one button on each landing, rather than the conventional two. Passengers press the button, but they are not guaranteed that the lift will be willing to go in the direction they wish to travel.

By inspecting the `button` module (figure 5), one finds that its variable `pressed` is set to false if the reset parameter is true; otherwise, if it was pressed before, it persists in that state; otherwise, it non-deterministically becomes true or

[4] Recall that the parameters are treated in a call-by-name fashion.

```
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.

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.

5.1.1 Properties for the basic lift system.

Before any features are added, we may use SMV to check basic properties of the lift system. For example, the following CTL[5] specification in the module

[5] 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))
```

```
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. 4. The SMV code for the module `lift` in the lift system.

`main` is satisfied: *pressing a landing button guarantees that the lift will arrive at that landing and open its doors*, i.e.:

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

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

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

These are some properties that we have verified for the base lift system and for its extensions with features. The results of our verifications are summarised in table 1. (The numbers in the table refer to the numbering in this list.)

(1) Pressing a landing button guarantees that the lift will arrive at that landing and open its doors:
    `AG (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 (liftBut`i`.pressed -> AF (floor=`i` & door=open))`
(3) The lift will not change its direction while there are calls in the direction it is travelling.
    One formula for upwards travel,
    `AG` $\forall i < j$. `(floor=`i` & liftBut`j`.pressed & direction=up`
              `-> A[direction=up U floor=`j`])`
    ... and one formula for downwards travel, for $i > j$:
    `AG` $\forall i > j$. `(floor=`i` & liftBut`j`.pressed & direction=down`
              `-> A[direction=up 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 (floor=`i` & door=closed & idle`
        `-> EG (floor=`i` & door=closed))`
    (The first formula states that the lift can actually get into a state satisfying the premise of the second formula.)
(6) The lift may stop at floors 2, 3, and 4 for landing calls when travelling upwards:
    $\forall i \in \{2,3,4\}$. `!AG ((floor=`i` & !liftBut`i`.pressed`
                   `& direction=up) -> door=closed)`
(7) The lift may stop at floors 2, 3, and 4 for landing calls when travelling downwards:
    $\forall i \in \{2,3,4\}$. `!AG ((floor=`i` & !liftBut`i`.pressed`
                   `& direction=down) -> door=closed)`

One can think of many more properties to check for a lift system. For this paper, we have omitted all safety properties, and have concentrated on a selection of properties that are characteristic of the SBCC algorithm, namely those that concern guarantee of service (or absence thereof).

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. 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.

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 6. The parking feature introduces the specification ((12) in table 1)

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

which says that the lift will not remain idle indefinitely at any floor other than floor 1. (In figure 6 we give only the instance for $i = 4$.)

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

5.2.1 Properties for the featured lift system.

In addition to the generic properties for the base system, we check some requirements for each feature. The results for these properties can also be found in table 1. (Again the numbering in the table corresponds to the numbers in

```
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

this list.)

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

(8) Empty:
The lift will not arrive empty at a floor unless the button on that landing was pressed.
`AG (lift.floor=`i` & lift.door=open & lift.empty`
`-> landingBut`i`.pressed)`

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

(10) Overloaded: (in `MODULE lift`)
The doors of the lift cannot be closed when the lift is overloaded.
`!EF (overload & door=closed)`

(11) Overloaded: (in `MODULE lift`)
The lift will not move while it is overloaded.
`AG (floor=`i` & overload -> A[ floor=`i` W !overload ])`

(12) Parking: (in `MODULE lift`)
The lift will not remain idle indefinitely at any floor other than floor 1.
`AG` $\forall i \neq 1.$ `!EG(floor=`i` & door=closed)`

(13) Lift-$\frac{2}{3}$-full: (in `MODULE lift`)

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

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

Our method provides a framework to plug these different features into the lift system, and by examining the result, to witness feature interactions. The SFI tool integrates one or more of the features, in a given order, into the base system. The result of our experimentation with the features for the lift system is summarised in table 1.

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

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

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

Table 1
Feature interactions for the lift system

	Property (see sections 5.1.1 and 5.2.1)													
Feature(s)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
no features	√	√	√	√	√	√	√	—	—	—	—	—	—	—
Empty	√	×	×	√	√	√	√	√	√	—	—	—	—	—
Overloaded	×	×	×	√	√	√	√	—	—	√	√	—	—	—
Parking	√	√	√	√	×	√	√	—	—	—	—	√	—	—
Lift-$\frac{2}{3}$-full	×	√	√	√	√	√	√	—	—	—	—	—	√	—
Exec. Floor	×	×	√	√	√	√	√	—	—	—	—	—	—	√
Overloaded * Empty	×	×	×	√	√	√	√	×	×	√	√	—	—	—
Parking * Empty	√	×	×	√	×	√	√	√	√	—	—	√	—	—
Lift-$\frac{2}{3}$-full * Empty	×	×	×	√	√	√	√	√	√	—	—	—	×	—
Exec. Floor * Empty	×	×	×	√	√	√	√	√	×	—	—	—	—	√
Parking * Overloaded	×	×	×	√	×	√	√	—	—	√	√	√	—	—
Lift-$\frac{2}{3}$-full * Overloaded	×	×	×	√	√	√	√	—	—	√	√	—	×	—
Exec. Floor * Overloaded	×	×	×	√	√	√	√	—	—	√	√	—	—	×
Lift-$\frac{2}{3}$-full * Parking	×	√	√	√	×	√	√	—	—	—	—	√	√	—
Exec. Floor * Parking	×	×	√	√	√	√	√	—	—	—	—	√	—	√
Exec. Floor + Lift-$\frac{2}{3}$-full	×	×	√	√	√	√	√	—	—	—	—	—	√	×
Lift-$\frac{2}{3}$-full + Exec. Floor	×	×	√	√	×	√	√	—	—	—	—	—	×	×

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

We can see that most feature interferences are of type I or II (*cf.* page 11), respectively, depending on the order of integration. Only combination Lift-$\frac{2}{3}$-full + Executive Floor produces a type III interaction. As mentioned above these features also exhibit a type IV interaction.

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

The reason for the second violation of a property of "Empty" by integrating "Overloaded" is quite different. Here the violation stems as much from the way we coded the property in CTL (`AG (lift.floor=`i` & lift.door=open & lift.empty -> landingBut`i`.pressed)`) as from the way the system and the features were coded. Essentially, in the base system, the lift would never stay at the same floor with its doors open for more than one step; and the buttons are reset when in that step. (*Cf.* property 5 in section 5.1.) With the "Overloaded" feature however it can happen that the lift is forced to keep its doors open – the premise of the implication holds, but the button has been reset (`landingBut`i`.pressed` = `false`).

We see that the violation occurs when both the flag `overload` and `empty` are true.[6] Obviously, in reality a lift can never be overloaded and empty at the same time, but our verification software and the feature integrator cannot know that. One possible solution would be to alter the features to take account of this constraint. However, this would contravene the modularity and independence of the features, so the best solution is to design another feature that implements the constraint, by either setting `overload` to false when `empty` is true or vice versa.

6 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 while her call is on hold.

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.

[6] The trace that SMV produces demonstrates this. A little reasoning shows that an interference is inevitable when `overload` = `empty` = 1.

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.

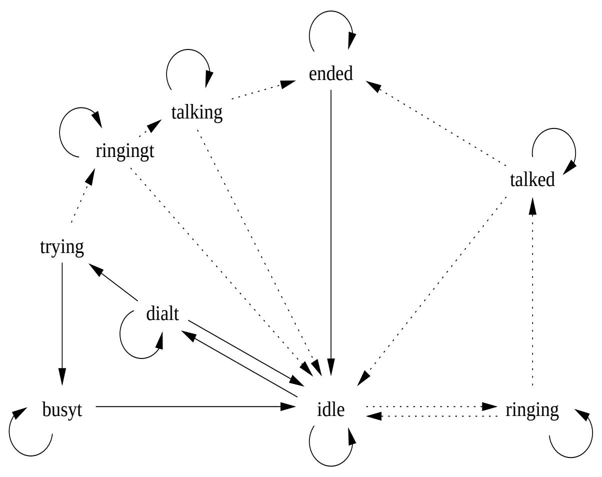
Originating Call Screening (OCS) This feature inhibits calls from the subscriber's phone to any number from a set chosen by the subscriber. Any attempt to ring such a number will yield an announcement.

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, without dialling the number.

6.1 The base system (POTS)

We have built an SMV description of a network of four synchronous phones. The behaviour of each phone is given by the finite automaton shown in figure 7, plus one variable, `dialled`, for each phone which indicates the phone to which it is connected (or to which it is trying to connect). Initially the phone is in state `idle`; from there, it may move to `ringing` (if someone rings it) or to `dialt` (if someone lifts the handset). `Dialt`, `ringingt`, and `busyt` abbreviate dial-tone, ringing-tone, and busy-tone, respectively. `Talking` represents the state where the phone is connected in a conversation which it initiated, while `talked` means that the conversation was initiated by someone else. `Ended` means that the party to which the phone was connected has hung up.

The variable `dialled` determines the other copy of the phone automaton with which these transitions have to synchronise. User input is simulated by non-determinism: the number to be dialled is non-deterministically chosen, and when there is more than one transition from a state, one is chosen non-deterministically. If a transition has to synchronise with a transition in another phone (indicated by a dotted line in the diagram), it can only be chosen if the other phone chooses the corresponding transition. In detail, the transitions are synchronised as follows:



(Dotted lines indicate synchronising transitions.)

Fig. 7. The automaton for a single phone.

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

Part of the code for the phone module can be seen in figures 8 and 9. In this piece of code one can also see how the synchronisation mechanism helps to avoid the race condition arising when several phones try to contact the same line at the same time. (In SMV we do this by using the `next` operator on the right hand side of an assignment.)

As it turned out, this model quickly grew too 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 7), the left hand side represents the originating line, and the right hand side the terminating line, both including the states `idle` and `ended`. Additionally, each feature was only added to one of the (complete) phones. A positive side-effect of this differentiation is that one can distinguish the interactions according to how features are distributed over the system.

For the features we modelled, we argue that the reduced model still exhibits all possible interactions of two feature instances if we go through all relevant combinations.
First we argue that four (complete) phones are sufficient. Each feature deals with at most three parties, and each phone can only originate one call (we

```
MODULE phone (X,B,C,D,p) -- parameters: the 4 numbers, and the array of phones
                         --             X is our own number
VAR
  dialled : {0,1,2,3,4};
  st      : {idle,dialt,trying,busyt,ringingt,talking,ringing,talked,ended};

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

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

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

      st=dialt : {dialt,trying};

      st=busyt : {idle,busyt};

      st=trying :
        case
          dialled=B & p[B].st=idle
          & ((p[C].st=trying & p[C].dialled=B)->next(p[C].st)=busyt)
          & ((p[D].st=trying & p[D].dialled=B)->next(p[D].st)=busyt) :ringingt;
          dialled=C & p[C].st=idle
          & ((p[B].st=trying & p[B].dialled=C)->next(p[B].st)=busyt)
          & ((p[D].st=trying & p[D].dialled=C)->next(p[D].st)=busyt) :ringingt;
          dialled=D & p[D].st=idle
          & ((p[B].st=trying & p[B].dialled=D)->next(p[B].st)=busyt)
          & ((p[C].st=trying & p[C].dialled=D)->next(p[C].st)=busyt) :ringingt;
          1                                                           :busyt;
        esac;

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

Fig. 8. The SMV code for the phone system. (1/2)

did not model Three Way Calling). Therefore a second feature may be added on any type of phone: one that is affected by the first feature in some way, or one that is not connected to it in any way. This gives rise to all interesting behaviours.

The main premise for our reasoning is that the effects of a feature are localised, i.e. only those phones which participate in a call affected by an instance of the feature, exhibit altered behaviour. We will use the term *configuration* to describe such a set of phones. To parties outside a configuration, the phones within the configuration behave as usual.[7] A direct consequence of this as-

[7] Should this not be the case (wrt. the properties we check), it would be detected when we test for successful integration of the feature: the feature would affect the operation of the network as seen by third parties, that have nothing to do with the feature instance or its subscriber.

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

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

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

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


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

Fig. 9. The SMV code for the phone system. (2/2)

sumption is that we only need to look at overlapping configurations when checking combinations of features. In essence, we are taking some of the choice away from the model checker, and in turn we need to ensure that we test all relevant cases. Checking different ways of overlapping will in general happen through the model checking process.

For the features we modelled in our case study, configurations comprise one, two or three phones. Only Call Waiting affects three phones, since the Call Forwarding features operate by re-routing the call – the forwarding phone is cut out of the configuration as soon as it diverts the call, and we are left with two phones in a standard call situation. In fact, the forwarding phone only provides the number to forward to. It does not have any new transitions to deal with forwarding, that functionality resides entirely in the phone which originated the call.

From this it is clear that overlapping configurations in a system with only two feature instances (from the set of features we consider) contain at most four distinct phones.

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

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

6.2 Integrating features into the telephone system

As an illustration of the feature construct we show the Ring Back When Free feature in figure 10. When looking at this example the reader should keep in mind that this code was written with the goal to run it through a model checker – and that the syntax which SMV accepts is rather limited. So for efficiency reasons, RBWF will only store one number at a time, and we do not allow cancelling RBWF once it is activated, until a call between the subscribed phone and the phone with the stored number has been established.

The `REQUIRE` section states that the feature needs a `MODULE phone` with at least the named parameters, and within that module, variables `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-use` and `rbwf-number`, and defines which number to store in `rbwf-number`, and under which conditions RBWF may be activated (`rbwf-use=1`) and deactivated (`rbwf-use=0`).

Finally, in the `CHANGE` section we define how the new variables interact with those of the base system. For the RBWF feature, the `CHANGE` section states that when both the subscriber's phone and the phone whose number was stored are idle, the subscriber's phone should try to connect to the phone 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; in fact this could be implemented as another feature. It would, however, slow

[8] If both calls are incoming calls, the situation is symmetrical and there are only two distinct roles.

```
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
```

Fig. 10. The code for the Ring Back When Free feature

down the model checking significantly, as we would have to introduce another variable to indicate the special ringing.

Apart from the generic properties of the phone system listed in the next section (6.2.1), we also want to verify that the base system with the feature actually behaves as the feature specification demands. For example, in the case of RWBF we also require the following (omitted in figure 10):

- 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 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.
```
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)
```
- 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 the following section.)

6.2.1 Properties of the basic phone system.

These are the properties that we have verified for the base system. (Again we use the meta-notation introduced on page 13.)

To save space, we have omitted from table 3 some more technical properties, but also these rather intuitive properties of the base system:

- The correct phone will ring: if phone i is trying to contact phone j and consequently gets the ringing-tone, then phone j must be ringing.
```
AG ((ph[i].st=trying & ph[i].dialled=j)
        -> AX (ph[i].st=ringingt -> ph[j].st=ringing))
```
- 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))
```
- 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))
```

The results for the following properties are given in table 3:

(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 in the next step.

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

(4) 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)
```

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

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

(6) 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 6.)

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

6.2.2 Properties for the featured phone system.

We derived the following requirements from the description of the services that these features implement and verified them for the respective features. For lack of space we only give one or two properties for each feature and omit some of the more technical properties that we verified.

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

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

(8) Call Forwarding on Busy:
If the subscriber's phone is busy, incoming calls will terminate at the phone with the forwarding number. (Again, the forwarding number remains fixed.)

```
AG ∀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)))
```

(9) Call Waiting:
If there are two calls to the subscribers phone, exactly one party will hear the 'onhold'-message. (In other words, at most one party will hear the 'onhold'-message at any given time.)

```
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))
```

(10) 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)
```

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

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

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

```
AG ((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)
```

(13) 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}))
```

(14) 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}))
```

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

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

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

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.[9] 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 RBWF to POTS+CW, we found that that RBWF did not respect the specifications introduced for CW (Type II interaction): this combination of features violated a requirement for CW (property (9) in section 6.2.2). The violated property states, that when there are two callers to a CW subscriber, exactly one of them is on hold at any given time.

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

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

(1) `ph[1]` tries to ring `ph[4]` when `ph[4]` is busy, and `ph[1]` activates RBWF;
(2) `ph[1]` then calls `ph[2]` (successfully);

[9] The latter result was omitted from table 3.

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

Table 2
Interferences between features for the phone system

	CW	CFU	CFB	RBWF	RBWF[1]	TCS	OCS
CW	—	IV	IV	II, IV	√	IV	II[4]
CFU	I, IV	—	IV	√	√	√	√
CFB	I, II, IV	II, IV	—	√	√	II	II
RBWF	II, IV	√	√	—	√	√	√
RBWF[1]	√	√	√	√	—	√	√
TCS	II, IV	√	I	√	√	—	√
OCS	I[4]	√	I	√	√	√	—

Table 2 indicates interferences between features for the phone system. A tick denotes that there is no interference, i.e. that both features work correctly together and it does not matter in what order they are integrated. When that is not the case, the table gives the types of interaction that we observed, according to the classification in 4. The superscripted numbers have the same meanings as in Table 3 and are explained below.

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

It is obvious from Table 3 that the interactions of Call Forwarding features with Call Screening features we could detect were determined by our decision not to model call legs. Had we modelled call legs, the combination of a Call Forwarding feature with a Call Screening feature would have violated the Call Screening property, since the extra call leg would interfer with determining the originator and terminator of a call, which is essential for screening the call.

7 Conclusions

Our approach to the feature-interaction problem gives features the status of first-class citizens; we could think of this as *feature orientation.* In concrete terms, this means that features are compact textual units in a specification or program, and that they are as independent as possible of the base system description, and features are independent of one another. 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.

The feature construct is most useful when the base system is *not* written in a 'feature ready' way. When one is dealing with a 'feature ready' specification

Table 3
Feature interactions for the telephone system

	Property (see sections 6.2.1 and 6.2.2)													
Feature(s)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
POTS	√	√	√	√	√	√	—	—	—	—	—	—	—	—
CW	√	×	×	×	×	×	—	—	√	√	—	—	—	—
CFU	×	√	×	√	√	×	√	—	—	—	—	—	—	—
CFB	√	√	×	√	√	×	—	√	—	—	—	—	—	—
RBWF	√	√	√	√	√	√	—	—	—	—	√	√	—	—
TCS	×	√	√	√	√	√	—	—	—	—	—	—	√	—
OCS	×	√	√	√	√	√	—	—	—	—	—	—	—	√
CW + CFU	×	×	×	×	×	×	√	—	√	√	—	—	—	—
CFU + CW	×	×	×	×	×	×	√	—	√	×	—	—	—	—
CW + CFB	√	×	×	√	×	×	—	√	√	√	—	—	—	—
CFB + CW	√	×	×	√	×	×	—	×	√	×	—	—	—	—
CW + RBWF	√	×	×	×	×	×	—	—	×	√	√	√	—	—
RBWF + CW	√	×	×	×	×	×	—	—	√	√	×	√	—	—
CW * RBWF[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]	×	√	√	√	√	√	—	—	—	—	√	√	—	√

(e.g. the "Intelligent Network" architecture for telephony [15]), this specification already defines the entities which can be manipulated by features and interfaces for these manipulations. Moreover the integration process is dictated by the architecture. Hence, in such a context, a feature construct would merely provide a uniform notation for features but would not add further modularisation.

The specifier of a feature needs a good understanding of the base system in order to make the feature operate correctly since the features are quite dependent on the underlying system. However, when designing a feature, the developer does not need to know about all other features that can be added to the system. With a feature construct, feature integration and interaction detection are completely automatic. Interferences between features are detected by model checking, and illustrated with traces, which help the developer to resolve the interferences.

The combination of feature integration and model checking has proved to be very useful. However, like all model checking applications, it suffers from the state space explosion problem. To overcome the state space explosion we were forced to use a rather abstract model. Due to the level of abstraction we chose, we missed some anticipated interactions, while on the other hand detecting some spurious interactions.

Our experiences with SMV as the underlying language were mixed. While SMV's compact language made it easy to define the feature construct and feature integration and the feature construct proved easy to use, we did find it not expressive enough for some purposes. Especially the restrictions on the usage of arrays and the lack of primitives for synchronisation and communication were cumbersome. On the other hand, the simplicity of SMV resulted in a small and simple feature construct, much simpler than our proposed feature construct for Promela [23].

Choosing SMV also meant that we were committed to the verification of properties stated in a logic, rather than testing processes for bisimulation or refinement, or checking generic properties such as deadlock-freedom. On the one hand, property based verification has the drawback that one might miss relevant aspects of the system; on the other hand, for most systems there is no general property that is not subject to changes by features. (The only generally desirable properties are probably deadlock- and livelock-freedom.) As a further development, one might want to automatically deduce some interesting properties from the feature implementation, such as checks that a feature's triggering conditions can actually arise in the system. An extension of this would be to check for overlaps in the conditions of various features, but we haven't explored that direction, yet.

Another positive result of defining a feature construct and the process of feature integration is that it allows a formal analysis of the semantics of features. This is the subject of another paper ([20]).

We expect that our approach will benefit from advances in verification technology. Two developments look especially promising: (semi-)automatic abstraction techniques [7,24] and simulation techniques that exercise the "interesting" parts of the system with good coverage, as demonstrated in [9].

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] L. G. Bouma and Hugo Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands, May 1994. IOS Press.

[4] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications III*, Tokyo, Japan, October 1995. IOS Press.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a reimplementation of SMV. In B. Steffen and T. Margaria, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT-98)*, BRICS Notes Series, pages 25–31, Aalborg, 1998. Available from http://afrodite.itc.it:1024/~cimatti/.

[6] 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.

[7] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[8] P. Dini et al., editors. *Feature Interactions in Telecommunications and Distributed Systems IV*, Montreal, Canada, June 1997. IOS Press.

[9] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation: a synchronous point of view. In Kimbler and Bouma [17], pages 262–275.

[10] Formal Systems (Europe) Ltd, Oxford, UK. *Failures-Divergence Refinement*, Oct 1997.

[11] Nancy Griffeth, editor. *1st International Workshop on Feature Interactions in Telecommunications Software Systems*, St. Petersburg, Florida, USA, December 1992.

[12] C. A. R. Hoare. *Communication Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[13] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[14] M. R. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 1999.

[15] ITU-T. *Intelligent Network – ITU Recommendations Q.1200 series*, 1995.

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

[17] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*, Lund, Sweden, Sept 1998. IOS Press.

[18] K. McMillan. The SMV language. Available from www-cad.eecs.berkeley.edu/˜kenmcmil, June 1998.

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

[20] M. Plath and M. Ryan. The semantics of a feature construct for SMV: A case study in non-monotonic composition. Technical report, School of Computer Science, University of Birmingham, 1999. Available as `ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1999/CSR-99-10.ps.gz`.

[21] M. C. Plath and M. D. Ryan. Plug-and-play features. In Kimbler and Bouma [17], pages 150–164.

[22] M. C. Plath and M. D. Ryan. SFI: a feature integration tool. *Advances in Computer Science*, pages 201–216, 1999.

[23] Malte Plath and Mark Ryan. A feature construct for Promela. In *SPIN'98 – Proceedings of the 4th SPIN workshop*, Nov 1998. Available as `http://netlib.bell-labs.com/netlib/spin/ws98/plath.ps.gz`.

[24] A. Pnueli. Verification by finitary abstraction. In *SPIN'98 – Proceedings of the 4th SPIN workshop*, Nov 1998. Available from www.wisdom.weizmann.ac.il/˜amir/invited-talks.html.