Using SPIN for Feature Interaction Analysis – a Case Study

M. Calder and A. Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.

Abstract. We show how SPIN is applied to analyse the behaviour of a real software artifact – feature interaction in telecommunications services. We demonstrate how minimal abstraction and optimisation techniques can greatly reduce the cost of model-checking, and how analysis can be performed automatically using scripts.

Keywords

telecommunications services; Promela/SPIN; communicating processes; distributed systems; formal modelling; analysis and reasoning techniques; feature interaction

1 Introduction

In software development a *feature* is a component of additional functionality – additional to the main body of code. Typically, features are added incrementally, often by different developers. A consequence of adding features in this way is *feature interaction*, when one feature affects, or modifies, the behaviour of another feature. Although in many cases feature interaction is quite acceptable, even desirable, in other cases interactions lead to unpredictable and undesirable results. The problem is well known within the telecommunications (telecomms) services domain (for example, see [2]), though it exhibits in many other domains such as email and electronic point of sales.

Techniques to deal with feature interactions can be characterised as design time or run time, interaction detection and/or resolution. Here, we concentrate on detection at design time, resolution will be achieved through re-design.

When there is a proliferation of features, as in telecomms services, then automated detection techniques are essential. In this paper, we investigate the feasibility of using Promela and SPIN [14].

Our approach involves considering a given service (and features) at four different levels of abstraction: communicating finite state automata, temporal logic formulae, Promela specifications and labelled transition systems and Büchi automata. We make contributions at several levels, including

- a low level call service model in Promela that permits truly independent call control processes with asynchronous communication, asymmetric call control and a facility for adding features in a structured way,

- optimisation techniques for Promela which result in tractable state-spaces, thus overcoming classic state-explosion problems,
- interaction analysis of a basic call service with six features, involving four users with full functionality. There are two types of analysis, static and dynamic, the latter is completely automated, making extensive use of Perl scripts to generate the model-checking runs.

Related work is discussed below. The overall approach to interaction detection, and the role of SPIN, is given in section 2; section 3 contains an introduction to feature interaction analysis. Sections 4, 5 and 6 give an overview of the finite state automata, temporal properties, the Promela implementation of the basic call service, and optimisations. Sections 7 and 8 contain an overview of the features and their implementations. The interaction analysis is described in sections 10 and 11 and in section 12 we discuss how the Promela models and SPIN model-checking runs required for the analysis are automated. We conclude in section 13.

1.1 Related Work

Model-checking for feature interaction analysis has been investigated using SMV [20], Caesar [21], COSPAN [10] and SPIN [16]. In the last, the Promela model is extracted mechanically from call processing software code; no details of the model are given and so it is difficult to compare results. In [20], the authors are restricted to two subscribers of the service with full functionality (plus two users with half functionality), due to state-explosion problems. For similar reasons, call control is not independent. Nevertheless, we regard this as a benchmark paper and aim at least to demonstrate a similar set of properties within our context. In [10] features and the basic service are described only at an abstract level by temporal descriptions. State-explosion is avoided, but interactions arising from implementation detail, such as race conditions, cannot be detected. Our layered approach permits this, building on earlier work by the first author in [21], using process algebra. This too suffered from limitations of state-explosion and the lack of (explicit) asynchronous communication; these limitations motivated the current investigation using Promela and SPIN. Initial attempts to model the basic call service using Promela and SPIN are described in [4].

2 Approach

Our approach has two phases; in the first phase we consider only the basic call service, as depicted in figure 1(a). The aim of the first phase is to develop the right level of abstraction of the basic service and to ensure that we have effective reasoning techniques, before proceeding to add features.

Our starting point is the top and left hand side of figure 1(a): the automata and properties. Neither need be *complete* specifications; this is a virtue of the approach and, for example, allows us to avoid the frame problem. The Promela

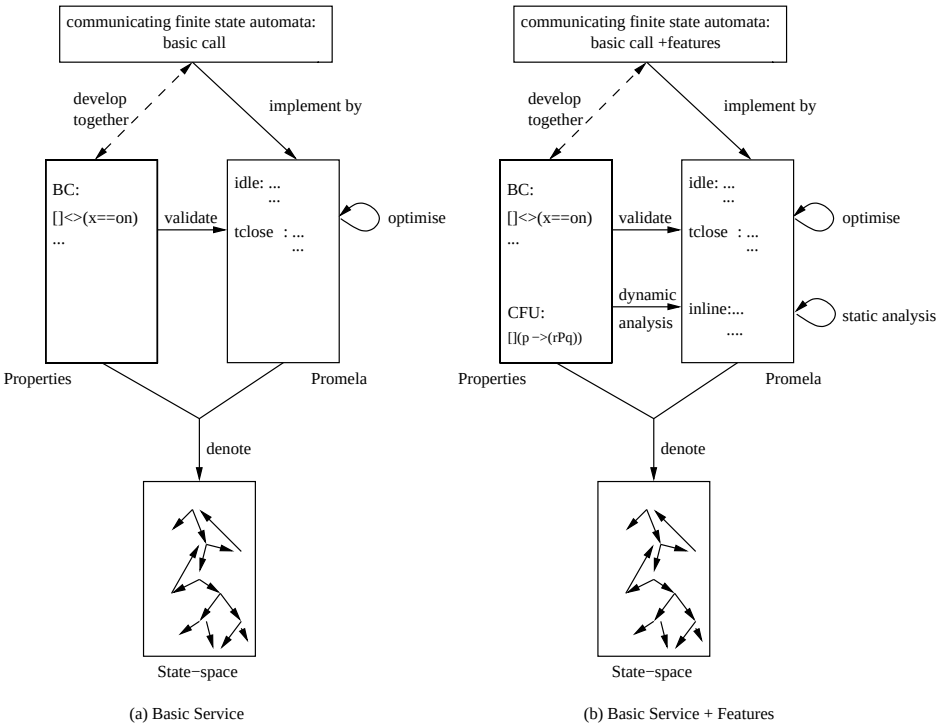

Fig. 1. Overall Approach

description on the rhs of figure 1(a) is regarded as the implementation; a crucial step therefore is validation of the implementation, i.e. checking satisfaction of the properties, using SPIN. Initial attempts fail, due to state-explosion, however, an examination of the underlying state-space (bottom of figure 1(a)) leads us to discover simple, but very effective optimisations.

The second phase, when we add features, is depicted in figure 1(b). Again, the starting point is finite state automata and properties. The Promela implementation is augmented with the new feature behaviour, primarily through the use of an *inline* function (see Section 9.1), and then validated. Interaction detection analysis takes two forms: *static* analysis, (syntactic) inspection of the Promela code, and *dynamic* analysis, reasoning over combinations of sets of logical formulae and configurations of the feature subscribers, using SPIN. The results of (either) analysis is interaction *detection*. The distinction between the two analyses is novel, we discuss this in more detail in Section 11.

3 Background– Features and Interactions

Control of the progress of calls is provided by a service at an exchange (a *stored program control* exchange). This software must respond to events such as handset

on or off hook, as well as sending control signals to devices and lines such as ringing tone or line engaged. A *service* is a collection of functionality that is usually self-sustaining. A *feature* is additional functionality, for example, a *call forwarding capability*, or *ring back when free*; a user is said to *subscribe* to a feature. When features are added to a basic service, there may be *interactions* (i.e. behavioural modifications) between both the features offered within that service, as well as with features offered in another service.

For example, if a user who subscribes to *call waiting* (CW) and *call forward when busy* (CFB) is engaged in a call, then what happens when there is a further incoming call? (Full details of all features mentioned here are given in section 7.) If the call is forwarded, then the CW feature is clearly compromised, and vice versa. In either case, the subscriber will not have his/her expectations met. This is an example of a single user, single component (SUSC) [5] interaction – the conflicting features are subscribed to by a single user. More subtle interactions can occur when more than one user/subscriber are involved, these are referred to as multiple user, multiple component (MUMC) interactions. Consider when user A subscribes to *originating call screening* (OCS), with user C on the screening list, and user B subscribes to CFB to user C. If A calls B, and the call is forwarded to C, as prescribed by B's CFB, then A's OCS is compromised. If the call is not forwarded, then we have the converse. These kind of interactions can be particularly difficult to detect (and resolve), since different features are activated at different stages of a the call.

Ideally, interactions are detected and resolved at service creation time, though this may not always be possible when third-party or legacy services are involved (for example, see [3]).

4 Basic Call Service

Figure 2 gives a diagrammatic representation of the automaton for the basic call service (following the IN (*Intelligent Networks*) model, distributed functional plane [17]).

States to the left of the idle state represent *terminating* behaviour, states to the right represent *originating* behaviour. Events observable by service subscribers label transitions: *user-initiated* events at the terminal device, such as (handset) on and (handset) off, are given in plain font, *network-initiated* events such as *unobt* and *engaged* are given in italics. Note that there are two "ring" events, *oring* and *tring*, for originating and terminating ring tone, respectively; call behaviour is asymmetric. Not all transitions are labelled.

The automata must communicate, in order to coordinate call set up and clear down. To implement communication, we associate a channel with each call process. Each channel has capacity for at most one message: a pair consisting of a channel name (the other party in the call) and a status bit (the status of the connection). Figure 3 describes how messages are interpreted.

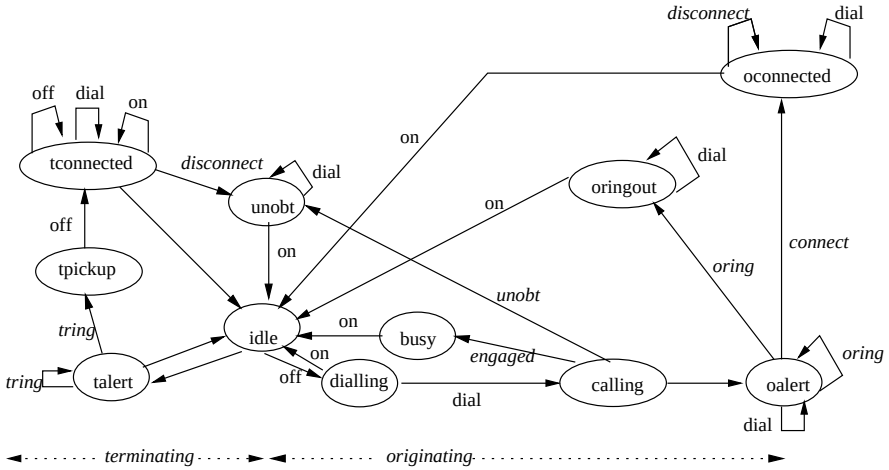

Fig. 2. Basic Call - States and Events

5 Basic Call Service Properties in LTL

Below, we give a set of temporal properties describing the behaviour of the basic call service. Before doing so, we make a few comments about our use of LTL in SPIN.

When using SPIN's LTL converter (or otherwise – we use the conversion tool of Etessami, [9]) it is possible to check whether a given property holds for *All Executions* or for *No Executions*. A universal quantifier is implicit at the beginning of all LTL formulas and so, to check an LTL property, it is natural to choose the *All Executions* option. However, we sometimes wish to check that a given property (p say) holds for *some state* along *some execution path* (or "p is possible"), we can do so by showing that "$\langle\rangle p$ holds for *No Executions*" is **not** true (via a never-claim violation), which is equivalent. In Property 1 below, we use the notation E to mean *for some path* in place of the usual implicit *for all*

Contents of Channel A	*Interpretation*
empty	A is free
(A,0)	A is engaged, but not connected
(B,0)	A is engaged, but not connected B is termnating party B is attempting connection
(B,1)	If channel B contains (A,1) then A and B are connected

Fig. 3. States of a Communication Channel in the Protocol

paths. Additionally, we can use SPIN to prove properties of the form "p is true in the next state relative to process i". (That is p is true after the next time that process i is active.) This is done via judicious use of SPIN's *_last* operator, details are omitted here. We use the shorthand $\circ_{proci}$ to mean the next global state in which process *proci* has made a local transition.

We adopt some notation of [6]: the operators $\mathcal{W}$ (*weak* until) and $\mathcal{P}$ (precedes), defined by $f\mathcal{W}g = []f \vee (fUg)$ and $f\mathcal{P}g = \neg(\neg fUg)$.

The LTL is given here alongside each property. This involves referring to variables (eg. *dialled* and *connect*) contained within the Promela code (an extract of which is given in section 6.1). We use symbols to denote predicates, for example "$[]p$ where p is $dialled[i] == i$". This provides a neater representation, and the LTL converter requires properties to be given in this way.

Property 1 *A connection between two users is possible.*

That is: $E\Diamond p$, where p is $connect[i].to[j] == 1$, for $i \neq j$.

Property 2 *If you dial yourself, then you receive the engaged tone before returning to the idle state.*

That is: $[](p \rightarrow ((\neg r)\mathcal{W}q))$ where p is $dialled[i] == i$, q is $network\_event[i] == engaged$ and r is $user[proci]@idle$.

Property 3 *Busy tone or ringing tone will directly (that is, the next time that the process is active) follow calling.*

That is: $[](p \rightarrow \circ_{proci} q)$ where p is $event[i] == call$ and q is $((network\_event[i] == engaged) \vee (network\_event[i] == oring))$.

Property 4 *The dialled number is the same as the number of the connection attempt.*

That is: $[](p \rightarrow q)$ where p is $dialled[i] == j$ and q is $partner[i] == chan\_name[j]$.

Property 5 *If you dial a busy number then either the busy line clears before a call is attempted, or you will hear the engaged tone before returning to the idle state.*

That is: $[]((p \wedge v \wedge t) \rightarrow (((\neg s)\mathcal{W}(w)) \vee ((\neg r)\mathcal{W}q)))$ where p is $dialled[i] == j$, v is $event[i] == dial$, t is $full(chan\_name[j])$, s is $event[i] == call$, w is $len(chan\_name[i]) == 0$, r is $user[proci]@idle$ and q is $network\_event[i] == engaged$, for $i \neq j$.

Note that the operator *len* is used to define w in preference to the function *empty* (or $nfull$). This is because SPIN disallows the use of the negation of these functions (and $\neg w$ arises within the never-claim).

Property 6 *You can not make a call without having just (that is, the last time that the process was active,) dialled a number.*

That is: $[](p \rightarrow q)$ where p is $user[proci]@calling$ and q is $event[i] == dial$.

6 Basic Call Service in Promela

6.1 Unoptimised Code

Each call process (see figure 2) is described in Promela as an instantiation of the (parameterised) proctype User declared thus:

```
proctype User (byte selfid;chan self)
```

Promela is a state-based formalism, rather than event-based. Therefore, we represent events by (their effect on) variables, and states (e.g. calling, dialling, etc.) by labels. Since each transition is implemented by several compound statements, we group these together as an *atomic* statement, concluding with a *goto*.

An example of the original (unoptimised) Promela code (as described in [4]) associated with the *idle*, *dialling*, *calling* and *oconnected* states and their outgoing transitions is given below. (For the full optimised code, contact the authors.) The global/local variables and parameters should be self-explanatory. We note in passing that any variable about which we intend to reason should not be updated more than once within any atomic statement; also d_steps, while more efficient than atomic steps, are not suitable here because they do not allow a process to jump to a label out of scope. There are numerous assertions within the code, particularly at points when entering a new (call) state, and when reading and writing to communication channels.

```
idle:
  atomic{
         assert(dev == on);
         assert(partner[selfid]==null);
     /* either attempt a call, or receive one */
         if
         :: empty(self)->event[selfid]=off;
            dev[selfid]=off;
            self!self,0;goto dialling
     /* no connection is being attempted, go offhook */
     /* and become originating party */
         :: full(self)-> self?<partner[selfid],messbit>;
     /* an incoming call */
            if
            ::full(partner[selfid])->
              partner[selfid]?<messchan,messbit>;
              if
              :: messchan == self /* call attempt still there */
                 ->messchan=null;messbit=0;goto talert
              :: else -> self?messchan,messbit;
     /* call attempt cancelled */
                 partner[selfid]=null;partnerid=6;
                 messchan=null;messbit=0;goto idle
              fi
            ::empty(partner[selfid])->
              self?messchan,messbit;
     /* call attempt cancelled */
              partner[selfid]=null;partnerid=6;
              messchan=null; messbit=0; goto idle
            fi
       fi};


dialling:
  atomic{
```

```
         assert(dev == off);assert(full(self));
         assert(partner[selfid]==null);
      /* dial or go onhook */
         if
         :: event[selfid]=dial;
      /* dial and then nondeterministic choice of called party */
            if
            :: partner[selfid] = zero;dialled[selfid] = 0;
               partnerid=0
            :: partner[selfid] = one;dialled[selfid] = 1;
               partnerid=1
            :: partner[selfid] = two;dialled[selfid] = 2;
               partnerid=2
            :: partner[selfid] = three;dialled[selfid] = 3;
               partnerid=3
            :: partnerid= 7;
            fi
         :: event[selfid]=on; dev[selfid]=on;
            self?messchan,messbit;assert(messchan==self);
            messchan=null;messbit=0;goto idle
     /*go onhook, without dialling */
         fi};

calling:/* check number called and process */
  atomic{
         event[selfid]=call;
         assert(dev == off);assert(full(self));
         if
         :: partnerid==7->goto unobtainable
         :: partner[selfid] == self -> goto busy
    /* invalid partner */
         :: ((partner[selfid]!=self)&&(partnerid!=7)) ->
            if
            :: empty(partner[selfid])->partner[selfid]!self,0;
               self?messchan,messbit;
               self!partner[selfid],0;goto oalert
    /* valid partner, write token to partner's channel*/
            :: full(partner[selfid]) -> goto busy
    /* valid partner but engaged */
            fi
         fi};

oconnected:
  atomic{
         assert(full(self));assert(full(partner[selfid]));
  /* connection established */
         connect[selfid].to[partnerid] = 1;
         goto oclose};
```

Any number of call processes can be run concurrently. For example, assuming the global communication channels `zero`, `one`, etc. a network of four call processes is given by:

```
atomic{run User(0,zero);run User(1,one); run User(2,two);run User(3,three)}
```

6.2 Options and Optimisation

Initial attempts to validate the properties against a network of four call processes fail because of state-explosion. In this section we examine the causes, the applicability of standard solutions and how the the Promela code can be transformed to optimise the state-space.

SPIN Options The default Partial order reduction (POR) option was applied throughout, but did not reduce the size of the state-space sufficiently. This is due to the scarcity of statically defined "safe" operations (see [15]) in our model. Any assignments to local variables are embedded in large atomic statements that are not safe. Furthermore the use of non-destructive read operations (to test the contents of a channel) prevents the assignment of exclusive read/send status to channels. Such a test is crucial: often behaviour depends on the exact contents of a channel.

States can be compressed using *minimised automaton encoding* (MA) or *compression* (COM). When using the former, it is necessary to define the maximum size of the state-vector, which of course implies that one has searched the entire space. However one can often find a reasonable value by choosing the (uncompressed) value reported from a preliminary verification with a deliberate assertion violation. While MA and COM together give a significant memory reduction, the trade-off in terms of time was simply unacceptable.

Other Optimisations A simple but stunningly effective way to reduce the state-space is to ensure that each visit to a *call* state is indeed a visit to the same underlying Promela state. This means that as many variables as possible should be initialised and then reset to their initial value (reinitialised) within Promela loops. For example, in virtually every call state it is possible to return to *idle*. An admirable reduction is made if variables such as `messchan` and `messbit` are initialised before the first visit to this label (*call* state), and then reinitialised before subsequent visits. This is so that global states that were previously *distinguished* (due to different values of these variables at different visits to the *idle* call state) are now *identified*.

The largest reduction is to be found when such variables are routinely reset before progressing to the next *call* state. Unfortunately, this is not always possible, as it would result in variables *about which we wish to reason* being updated more than once within an atomic statement (as discussed in section 6.1). However, there is a solution: add a further state where variables are reinitialised. For example, we have added a new state *preidle*, where the variables *network_event* and *event* are reinitialised, before progression to *idle*. Therefore every occurrence of *goto idle* becomes *goto preidle*.

We note that although the (default) data-flow optimisation option available with SPIN attempts to reinitialise variables automatically, we have found that this option actually *increases* the size of the state-space of our model. This is due to the initial values of our variables often being non-zero (when they are of type mtype for example). SPIN's data-flow optimisation always resets variables to zero. Therefore we *must* switch this option off, and reinitialise our variables manually.

The size of the state-space can be greatly reduced if any reference to (update of) a global variable which is not needed for verification, is commented out. Furthermore, by including all references to *all* of the *event* variables (say) when any such variable is needed for verification (see for example Property 3), the size of the state-space can be increased by an unnecessarily large amount. For

example, to prove that Property 3 holds for $user[i]$, we are only interested in the value of $event[i]$, not of $event[j]$ where $i \neq j$. The latter do not need to be updated. Thus an inline function, $event\_action(eventq)$ has been introduced to enable the *update of specific variables*. That is, it allows us to update the value of $event[i]$ to the value $eventq$, and leave the other event variables set to their default value. So, for example, if $i = 0$, the event_action inline becomes:

```
inline event_action (eventq)
{
 if
 ::selfid==0->event[selfid]=eventq
 ::selfid!=0->skip
 fi
}
```

Any reference to this inline definition is merely commented out when no *event* variables are needed for verification. (Another inline function is included to handle the *network_event* variables in the same way.)

We note that this reduction is not implemented in SPIN. SPIN does, however, issue a warning "variable never used" in situations where such a reduction would be beneficial.

These transformations not only lead to a *dramatic* reduction of the underlying state-space, the search depth required was reduced to 10 percent of the initial value, but they do not involve abstraction away from the original model. On the contrary, if anything, they could be said to reduce the level of abstraction.

6.3 Basic Call Service Validation

It was possible to verify all six properties well within our 1.5 Gbyte memory limit. State compression was used throughout. The verification of property 3 took the longest (21 mins) and a greater search-depth was reached in this case. This is partially due to the fact that both the *event* and *network_event* variables for the process under consideration had to be included for this property. In addition, the use of the *_last* operator precludes the use of partial order reduction, which could have helped to reduce the complexity in this case.

7 Features

Now that the state-space is tractable, we can commence the second phase: adding a number of features to the basic service.

7.1 Features

The set of features that we have added include:

- **CFU – call forward unconditional** All calls to the subscriber's phone are diverted to another phone.
- **CFB – call forward when busy** All calls to the subscriber's phone are diverted to another phone, if and when the subscriber is busy.

- **OCS – originating call screening** All calls by the subscriber to numbers on a predefined list are inhibited. Assume that the list for user x does not contain x.
- **ODS – originating dial screening**. The dialling of numbers on a predefined list by the subscriber is inhibited. Assume that the list for user x does not contain x.
- **TCS – terminating call screening** Calls to the subscriber from any number on a predefined list are inhibited. Assume that the list for user x does not contain x.
- **RBWF – ring back when free** The subscriber has the option to call the last recorded caller to his/her phone.

Two further features that are straightforward to implement are originating call behaviour (e.g. a pay phone) and terminating call behaviour (e.g. a teen line). However we give no details of such features here.

We do not give automata for all the features, but in figure 4 we give the additional behaviour prescribed by the RBWF feature. Notice that this feature introduces a new call state (namely *ringback*); it is the only feature to do so.

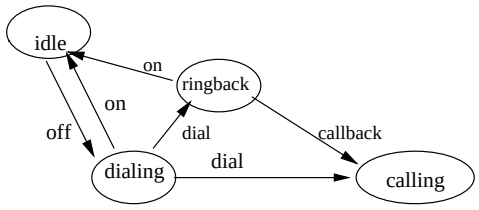

Fig. 4. Part of the Finite State Automaton for RBWF

8 Temporal Properties for Features

The properties for features are more difficult to express than those for the basic service. In order to accurately reflect the behaviour of each feature great attention must be paid to the *scope* of each property (within the corresponding LTL formula). For example, in property 8, it is essential that (for the CFB feature to be invoked) the forwarding party has a full communication channel *whilst the dialling party is in the dialling state*. This can only be expressed by stating that the forwarding party must have a full channel continuously between two states, the first of which must occur *before* the dialling party enters the dialling state, and the second *after* the dialling party emerges from the dialling state.

The values of the variables i, j and k depend on the *particular pair of features* and the corresponding *property* that is being analysed. These variables are therefore updated prior to each verification either manually (by editing the Promela

code directly), or automatically during the running of a model-generating script (see section 12).

Property 7 – CFU Assume that user j forwards to k.
If user i rings user j then a connection between i and k will be attempted before user i hangs up.
That is: $[](p \rightarrow (r \mathcal{P} q))$, where p is $dialled[i] == j$, r is $partner[i] == chan\_name[k]$, and q is $dev[i] == on$.

Property 8 – CFB Assume that user j forwards to k.
If user i rings user j when j is busy then a connection between i and k will be attempted before user i hangs up.
That is: $[](((u \wedge t) \wedge ((u \wedge t) U ((\neg u) \wedge t \wedge p))) \rightarrow (r \mathcal{P} q))$, where p is $dialled[i] == j$, t is $full(chan\_name[j])$, r is $partner[i] == chan\_name[k]$, u is $user[proci]@dialling$ and q is $dev[i] == on$.

Property 9 – OCS Assume that user i has user j on its screening list.
No connection from user i to user j is possible.
That is: $[](\neg p)$, where p is $connect[i].to[j] == 1$.

Property 10 – ODS Assume that user i has user j on its screening list.
User i may not dial user j .
That is: $[](\neg p)$, where p is $dialled[i] == j$.

Property 11 – TCS Assume that user i has user j on its screening list.
No connection from user j to user i is possible.
That is: $[](\neg p)$, where p is $connect[j].to[i] == 1$.

Property 12 – RBWF Assume that user j has automatic call back.
It is possible for an attempted call from i to j to eventually result in a successful call from j to i (without j ever dialling i) .

That is: $E(\diamond((p \wedge t \wedge \diamond q) \wedge (r \mathcal{P} q))$, where p is $dialled[i] = j$, q is $dialled[j] = i$, r is $connect[j].to[i] == 1$ and t is $event[i] == call$.

9 The Features in Promela

Relevant changes that need to be made to the Promela model are given below. Before this, we make a few observations:

- To implement the features we have included a "feature_lookup" function (see below) that implements the features and computes the transitive closure of the forwarding relations (when such features apply to the same call state).
- We distinguish between call and dial screening; the former means a call between user A and B is prohibited, regardless of whether or not A actually dialled B, the latter means that if A dials B, then the call cannot proceed, but they might become connected by some other means. The latter case might be desirable if screening is motivated by billing. For example, if user A dials C (a local leg) and C forwards calls to B (a trunk leg) then A would only pay for the local leg.

- Currently we restrict the size of the lists of screened callers (relating to the OCS, ODS and TCS features) to one. That is, we assume that it is impossible for a single user to subscribe to two of the *same* screening feature. This is sufficient to demonstrate some feature interactions, and limits the size of the state-space.
- The addition of RBWF, while straightforward, increases the complexity of the underlying state-space greatly. This is due both to the addition of the new *ringback* state and to the fact that it involves recording (in a structure indexed by call processes) the last connection attempt. The issue is not just that there is a new global variable, but that *call* states that were previously identified are now distinguished by the contents of that record (see discussion about variable reinitialisation in section 6.2).
- To ensure that all variables are initialised, we use 6 as a default value. This is particularly useful when a user does not subscribe to a particular feature. The value 7 is used to denote both an unobtainable number (e.g. an incorrect number) and to denote the "button press" in RBWF. We do not use an additional value for the latter, so as not to increase the state space.

9.1 Implementation of features: the *feature_lookup* inline

In order to enable us to add features easily, all of the code relating to *feature behaviour* is now included within an *inline* definition. The *feature_lookup* inline is defined as follows:

```
inline feature_lookup(part_chan,part_id,st)
{
 do
 ::((st==st_dial)&&(ODS[selfid]==part_id))->st=st_unobt
 ::((st==st_dial)&&(RBWF[selfid]==1)&&(part_id==7))->st=st_rback
 ::((part_id!=7)&&(st==st_dial)&&(CFU[part_id]!=6))
          ->part_id=CFU[part_id];part_chan=chan_name[part_id]
 ::((part_id!=7)&&(st==st_dial)&&(CFB[part_id]!=6)&&(len(part_chan)>0))
          ->part_id=CFB[part_id];part_chan=chan_name[part_id]
 ::((st==st_call)&&(OCS[selfid]==part_id))->st=st_unobt
 ::((st==st_call)&&(TCS[part_id]==selfid))->st=st_unobt
 ::else->break
 od
 }
```

The parameters *part_chan*, *part_id*, and *st* take the values of the current partner, partnerid and state of a user when a call to the the *feature_lookup* inline is made. Statements within *feature_lookup* pertaining to features that are not currently active are automatically commented out (see section 12).

We note that in some sense *feature_lookup* encapsulates centralised intelligence in the switch, as it has "knowledge" of the status of processes and data concerning feature configuration. While on the one hand one might argue that this is against the spirit of an *IN* switch, on the other hand we maintain that MUMC feature interactions simply cannot be detected in a completely distributed architecture.

9.2 Feature Validation

Each feature was validated (via SPIN verification) against the appropriate set of properties (1–12). For brevity, we do not give details here.

10 Static Analysis

Static analysis is an analysis of the *structure* of the feature descriptions, i.e. an examination of the *syntax*. Specifically, we look for *overlapping* guards (two or more guards which evaluate to true, under an assignment to variables) with diverging consequences. A more operational explanation is the detection of shared *triggers* of features. Because we have collected additional feature behaviour together within the inline *feature_lookup*, we need only consider overlapping guards within this function. If there is an overlap, and the consequences diverge, then we have non-determinism and hence a potential interaction.

For example, consider the following overlap between CFU and CFB:

```
::((part_id!=7)&&(st==st_dial)&&(CFU[part_id]!=6))
      ->part_id=CFU[part_id]; part_chan=chan_name[part_id]
::((part_id!=7)&&(st==st_dial)&&(CFB[part_id]!=6)&&(len(part_chan)>0))
      ->part_id=CFB[part_id];part_chan=chan_name[part_id]
```

The overlap occurs under the assignment $st = st\_dial$, $CFU[part\_id] = x$, $len(part\_chan) > 0$, and $CFB[part\_id] = y$ where $x, y \neq 6$. When $x \neq y$, the first consequent assigns x to $part\_id$, the second assigns y to $part\_id$. These are clearly divergent, and so we have found an interaction.

SUSC and MUMC interactions are distinguished by considering the roles of $part\_id$ and $selfid$ as indices. If the same index is used for the feature subscription, e.g. $CFU[part\_id]$ and $CFB[part\_id]$, then the interaction is SUSC, if different indices are used, it is MUMC. In this example, the interaction is clearly SUSC.

An overlap is not always possible. For example, consider the first two choices:

```
::((st==st_dial)&&(ODS[selfid]==part_id))
       ->st=st_unobt
::((st==st_dial)&&(RBWF[selfid]==1)&&(part_id==7))
       ->st=st_rback
```

As 7 is not a valid number to be in a screening list there is no overlap and hence no interaction.

In all, there are 7 pairs to consider (4 clauses for st_dial, leading to 6 pairs, and two clauses for st_call, leading to one pair). Results of the static analysis are given in the tables of figure 5. A $\surd$ indicates an interaction whereas a $\times$ indicates none. The tables are symmetric.

Static analysis is a very simple yet very effective mechanism for finding some interactions – those which arise from new non-determinism. It is based on equational reasoning techniques and the process of finding overlapping guards (known as superposition) can be automated. The process of considering whether the consequent statements are divergent is more difficult and a complete solution would require a thorough axiomatic description of the Promela language. However, it

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	–	✓	×	×	×	×
CFB	✓	–	×	×	×	×
OCS	×	×	–	×	×	×
ODS	×	×	×	–	×	×
TCS	×	×	×	×	–	×
RBWF	×	×	×	×	–	–

(a) SUSC

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	–	×	×	✓	×	×
CFB	×	–	×	✓	×	×
OCS	×	×	–	×	×	×
ODS	✓	✓	×	–	×	×
TCS	×	×	×	×	–	×
RBWF	×	×	×	×	×	–

(b) MUMC

Fig. 5. Feature Interaction Results - Static Analysis

would be possible to automate a relatively effective approach based on simple assignment. For the purposes of this paper, we rely on manual inspection of the function *feature_lookup*. In any case, we note that the ease and contribution of static analysis depends very much on the structure of the specification.

We now turn our attention to a *dynamic* form of analysis.

11 Dynamic Analysis

Dynamic analysis depends upon logical properties that are satisfied (or not) by pairs of users subscribing to combinations of features.

Consider two users, $u1$ and $u2$. Then $u1_{f_i} \cup u2_{f_j}$ is the *configuration*, or *scenario*, in which $u1$ subscribes to feature f_i and $u2$ subscribe to feature f_j. Two features f_i and f_j *interact* if a property that holds for f_i alone, no longer holds in the presence of another feature f_j. More formally stated: for a property ϕ, we have $u1_{f_i} \models \phi$ but $u1_{f_i} \cup u2_{f_j} \not\models \phi$. When $u1 == u2$, then the interaction is SUSC, otherwise it is MUMC.

Note that the analysis is *pairwise*, known as 2-way interactions. While at first sight this may be limiting, empirical evidence suggests there is little motivation to generalise, 3-way interactions that are not detectable as a 2-way interaction are exceedingly rare [19].

An initial approach is to consider *any* property above as a candidate for ϕ. However, it is easy to see that in this case all features interact. A more selective approach is required: we consider only the properties associated with the features under examination, i.e. for features f_i and f_j, consider only properties ϕ_i and ϕ_j. An SUSC (MUMC) interaction between f_i and f_j, resulting from a violation of property ϕ_i is written $(f_i, f_j)_S$ $((f_i, f_j)_M)$.

11.1 Dynamic Analysis – Feature Interaction results

The tables in figure 6 gives the interactions found for pairs of features in both the SUSC case and the MUMC case. A $\surd$ in the row labelled by feature f_i means that the property ϕ_i is violated whereas a $\times$ indicates that no such violation has occurred. Two features f_i and f_j interact if and only if there is a $\surd$ in position

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	–	√	×	×	×	×
CFB	√	–	×	×	×	×
OCS	×	×	–	×	×	×
ODS	×	×	×	–	×	×
TCS	×	×	×	×	–	×
RBWF	√	×	√	√	√	–

(a) SUSC

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	√	√	×	×	×	×
CFB	√	√	×	×	×	×
OCS	×	×	×	×	×	×
ODS	√	√	×	×	×	×
TCS	×	×	×	×	×	×
RBWF	×	×	√	√	√	×

(b) MUMC

Fig. 6. Feature Interaction Results - Dynamic Analysis

(f_i, f_j) and/or a $\surd$ in position (f_j, f_i). BC is excluded as every feature interacts with it in some way.

New SUSC interactions are detected by the dynamic analysis, namely those associated with the RBWF feature. For example, there is an $(RBWF, CFU)_S$ interaction because the CFU feature prevents the *record* variable pertaining to the subscriber being set to a non-default value. Therefore the subscriber is unable to perform a ring-back.

The tables are not symmetric. For example, there is an $(ODS, CFU)_M$ interaction, but not a $(CFU, ODS)_M$ interaction. To understand why, observe that static analysis detects an MUMC interaction under the assignment $ODS[0] = 1$, and $CFU[1] = 2$. Dynamic analysis also detects an interaction violation – indeed our analysis script (see section 12) generates exactly this scenario: an $(ODS, CFU)_M$ interaction with $i = 0$ and $j = 1$ (i.e. user 0 rings user 1). Consider those computations where $feature\_lookup$ takes the ODS branch. One could understand this as ODS having precedence. There is no interaction in this case: both property 7 and property 10 are satisfied. However, there is a computation where the CFU branch is taken; in this case CFU has precedence and property 10 is violated because user 0 has *dialled* user 1 – before the call is forwarded to user 2 (although clearly property 7 is satisfied). Often, understanding why and how a property is violated will give the designer strong hints as to how to resolve an interaction.

The interactions uncovered by dynamic analysis depend very much on the *properties* and how the features are *modelled*. When the properties are *adequate*, we would expect every statically detected interaction to be detected dynamically, but not vice versa. This is borne out by our case-study. We may regard the static analysis step as a cheap method of uncovering some interactions, as well as providing an indication of whether or not we have a good set of behavioural properties. But, note that the properties are not complete descriptions, in particular they do not state what should *not* happen (i.e. the frame problem). For example, one might expect a $(CFU, TCS)_M$ interaction but this is not the case because although TCS will block the forwarded call, the *partner* variable will be set appropriately, thus satisfying property 7. Perhaps one should strengthen the property for CFU, to insist that the connection is made (rather than just setting *partner* appropriately). But it is not that simple, the forwarded party

may be engaged, or have a forwarded feature (or any other kind of feature); the possibilities are endless. Therefore, we consider the CFU property to be quite adequate.

12 Automatic Model Generation and Feature Interaction

Originally, before features were added to the basic call model, global variables were manually "turned off" (ie. commented out) or replaced by local variables when they are not needed for verification. The addition of features has led to even more variables requiring to be selectively turned on and off, and set to different values. For example if an *originating call screening* feature is selected the *orig_call_sreen* array has to be included and its elements set to the appropriate values. In addition the *feature_lookup* inline must be amended to include those lines pertaining to the originating call screening feature. If no *ring back when free* feature is chosen, the entire *ringback* call state must be commented out.

Making all of the necessary changes before every SPIN run was extremely time-consuming and error prone. Therefore, we now use a Perl script to enable us to perform these changes automatically. Specifically this enables us to generate, for any combination of features and properties, a model from a template file. Each generated model also includes a header containing information about which features and properties have been chosen in that particular case, which makes it easier to monitor model-checking runs.

Dynamic feature interaction analysis is combinatorially explosive: we must consider all pairs of features *and* combinations of suitable instantiations of the free variables i,j and k occurring in the properties. For example, for the SUSC case alone this gives 36 different scenarios (though not all are valid). To ease this burden and to speed up the process, a further Perl script is used to enable

- systematic selection of pairs of features and parameters i,j and k, and generation of corresponding model,
- automatic SPIN verification of model and recording of feature interaction results.

Note that scenarios leading to feature interactions *are* recorded. Depending on the property concerned, a report of 1 error (properties 7–11) or 0 errors (property 12) from the SPIN verification indicates an interaction. Once (if) an SUSC interaction is found the search for MUMC interactions commences. If an MUMC interaction is found the next pair of features is considered. The following example of output demonstrates the complete results for CFU and CFB with property 7.

```
/*The features are 1 and 2 */

/*New combination of features:CFU[0]=1 and CFB[0]=0 */
feature 2 is meaningless

/*New combination of features:CFU[0]=1 and CFB[0]=1 */
with property 7
with parameters 0,0 and 1 errors: 0
```

```
with parameters 1,0 and 1 errors: 0

with parameters 2,0 and 1 errors: 0

with parameters 3,0 and 1 errors: 0


/*New combination of features:CFU[0]=1 and CFB[0]=2 */
with property 7
with parameters 0,0 and 1 errors: 1 FEATURE INTERACTION: SUSC

/*New combination of features:CFU[0]=1 and CFB[1]=0 */
potential loop, test seperately

/*New combination of features:CFU[0]=1 and CFB[1]=1 */
feature 2 is meaningless

/*New combination of features:CFU[0]=1 and CFB[1]=2 */
with property 7
with parameters 0,0 and 1 errors: 1 FEATURE INTERACTION: MUMC
```

13 Conclusions and Future Directions

We have used Promela and SPIN to analyse the behaviour of a software artifact – feature interaction in a telecomms service. Our approach involves four different levels of abstraction: communicating finite state automata, temporal logic formulae, Promela specifications, and the underlying labelled transition systems and Büchi automata.

We have demonstrated the approach with an analysis of a basic call service with six features, involving four users with full functionality. There are two types of analysis, static and dynamic; the latter is completely automated, making extensive use of Perl scripts to generate the SPIN runs.

The distinction between static and dynamic analysis is novel; the latter is more comprehensive, but the former provides a simple yet effective initial step, and a check for the temporal properties upon which the latter depends.

State-explosion is a major concern in feature interaction analysis: understanding how a Promela model can be optimised, in order to generate tractable state-spaces, is important. We have outlined a simple but effective optimisation technique for Promela that does not abstract away from the system being modelled, on the contrary, it may be understood as reducing the gap between the Promela representation and the system under investigation. The technique involves reinitialising variables and results in a reduction of 90 per cent of the state-space. Thus, we overcome classic state-explosion problems and our interaction analysis results are considerably more extensive than those in [20].

Finally, we note that understanding why an interaction occurs can help the redesign process. For example, static analysis indicates shared triggers and dynamic analysis indicates in-built precedences between features, when the results of the analysis are not symmetric. Both can indicate how to alter precedences between features, in order to resolve interactions. How to do so in a structured way is a topic for further work.

Acknowledgements The authors thank Gerard Holzmann for his help and advice, and the Revelation project at Glasgow for computing resources. The second author was supported by a Daphne Jackson Fellowship from the EPSRC.

References

1. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press (Amsterdam), May 1994.
2. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems*, volume VI. IOS Press, Amsterdam, 2000.
3. M. Calder, E. Magill, and D. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *IEE Proceedings - Software*, 146(3):167–180, June 1999.
4. M. Calder and Alice Miller. Analysing a basic call protocol using Promela/XSpin. In *[13]*, pages 169–181, 1998.
5. E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for IN and beyond. In *[1]*, pages 1–23, May 1994.
6. Marsha Chechik and Dimitrie O. Paun. Events in property patterns. In *[7]*, pages 154–167, 1999.
7. D. Dams, R. Gerth, S. Leue, and M. Massink, editors. *Theoretical and Practical Aspects of Spin Model Checking: Proceedings of the 5th and 6th International Spin Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
8. P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV*. IOS Press (Amsterdam), June 1997.
9. K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *[11]*, pages 236–248, 1999.
10. A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In *[2]*, pages 179–192, May 2000.
11. Nicolas Halbwachs and Doron Peled, editors. *Proceedings of the eleventh International Conference on Computer-aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.
12. D. Hogrefe and S. Leue, editors. *Proceedings of the Seventh International Conference on Formal Description Techniques (FORTE '94)*, volume 6 of *International Federation For Information Processing*, Berne, Switzerland, October 1994. Kluwer Academic Publishers.
13. Gerard Holzmann, Elie Najm, and Ahmed Serhrouchni, editors. *Proceedings of the 4th Workshop on Automata Theoretic Verification with the Spin Model Checker (SPIN '98)*, Paris, France, November 1998.
14. Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
15. Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *[12]*, pages 197–211, 1994.
16. G.J. Holzmann and Margaret H. Smith. A practical method for the verification of event-driven software. In *Proceedings of the 1999 international conference on Software engineering (ICSE99)*, pages 597–607, Los Angeles, CA, USA, May 1999. ACM Press.
17. *IN Distributed Functional Plane Architecture*, recommmendation q.1204, ITU-T edition, March 1992.

18. K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.
19. M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In *[2]*, pages 311–325, May 2000.
20. M. Plath and M. Ryan. Plug-and-play features. In *[18]*, pages 150–164, 1998.
21. M. Thomas. Modelling and analysing user views of telecommunications services. In *[8]*, pages 168–182, 1997.