Model Checking for Feature Interaction Analysis: *controlling state-explosion*

Muffy Calder and Alice Miller[*]
Department of Computing Science
University of Glasgow
Glasgow, Scotland.

August 28, 2000

Abstract

keywords

model-checking; telecommunications services; Promela/Spin; distributed systems; feature interaction; formal methods.

1 Introduction

This paper considers some of the practical challenges that arise when using the model-checker Spin [7] to reason about a family of complex distributed systems; the application domain is feature interaction analysis in telecommunications services. Communication between call processes is asynchronous and mediated by a protocol. The systems being modelled are close to operational code and by their very nature, complex ones with very large underlying state-spaces. Considering a higher level of abstraction is not an option – it is important to be able to simulate, and reason about, behaviour at a lower level in order to detect and analyse interactions between features.

In modern telecomms systems, control of the progress of calls and connections is provided by software 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 the collection of functionality provided by a network operator and is usually self-sustaining. A *feature* is additional functionality, for example, a *call forwarding capability*, or *ring back when free*. When features are added to a

[*]Supported by a Daphne Jackson Fellowship Funded by the Engineering and Physical Sciences Research Council

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. Ideally, these interactions are detected and resolved at service creation time, though this may not always be possible when third-party or legacy services are involved (c.f. [1]). Regardless, good techniques for modelling and analysing services and features are required.

Model-checking for feature interaction analysis has been investigated by others. For example, SMV is employed in [10], Caesar in [11], and Spin in [8]. In the latter the Promela model is extracted mechanically from call processing software code, and no details of the model are given. Here, we extend the applicability and effectiveness of of model-checking in this problem domain in several ways by developing:

- a low level service model in Promela that permits truly independent call control processes with asynchronous communication, asymmetric call control (only an originating party can terminate a call), and includes a number of features,
- techniques for reducing potentially intractable state-spaces by transforming the Promela description without abstracting away from low level behaviour,
- techniques for checking temporal formulae in a distributed system with *relativised* temporal operators – operators that depend on constituent processes.

We believe that not only are our results interesting in themselves, but our experience provides useful guidance for others model-checking in this, and other, application domains.

The paper is organised as follows. A brief overview of Promela and Spin is given in the next section. Following that, we introduce the basic call service and its implementation in Promela. In section 4 we give a set of temporal properties that are important for behavioural analysis and discuss how they are best represented in the Spin/Promela paradigm. We evaluate initial attempts to verify those properties, the consequent optimisations made to the Promela code, and further verification results after optimisation. In section 5, we define a set of features and further properties to be considered. Following that, we validate the features against the appropriate properties and then perform the interaction analysis. Conclusions and directions for further work are in the final section; a complete listing of the Promela code is given in the Appendix.

2 Promela and Spin

Promela (*Pro*cess *me*ta *la*nguage ([7])) is an imperative C-like language for modelling communicating, concurrent processes. It includes constructs for non-determinism, asynchronous and synchronous communication, dynamic process

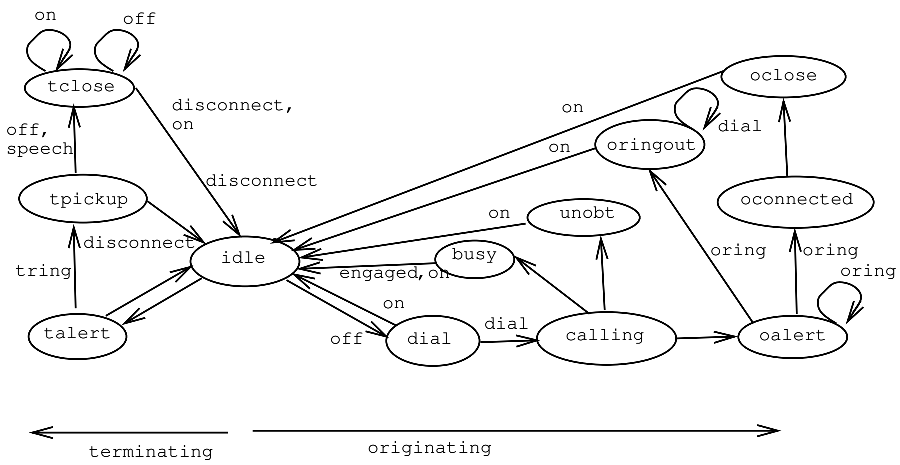

Figure 1: Basic Call Service: States and Events

creation, and mobile connections, i.e. communication channels can contain other communication channels. Thus, the language is very powerful and expressive. SPIN is a bespoke model-checker for Promela. Both were developed by G. Holzmann at AT&T (now Lucent Technologies) in the USA and are now used extensively throughout the USA and Europe for numerous telecommunications and reactive systems applications, both in academe and in industry. In addition to symbolic simulation, Spin provides several reasoning mechanisms: assertion checking, acceptance/progress states and cycles, and satisfaction of linear temporal logic (LTL) formulae.

3 Overview of the Basic Service

As with all distributed systems, there are many perspectives, or views of a telecommunications network, with varying levels of abstraction. Here, we have chosen to examine the "user" perspective of the service, following the IN (*Intelligent Networks*) model, distributed functional plane [5]. An overview of the basic call service is given by a state transition diagram in figure 1; *originating party* behaviour is to the right of the *idle* state, *terminating party* is to the left.

Transitions are labelled by *events*. These may be *user-initiated* events at the terminal device, such as (handset) *on*, (handset) *off*, or *network-initiated* events such as *unobt*, *engaged*. Note that there are two "ring" events, *oring* and *tring*, for originating and terminating ring tone, respectively, this reflects the fact that the ringing tone is indeed generated at each terminal device.

Call processes must communicate and respond to each other, this is usually achieved by means of a switch. Here, to coordinate communication, we associate a channel with each process. Each such 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). When a communication channel is empty, then that user is not connected to, or attempting to connect to, any other user. We will refer to this situation as the user is *not engaged in a call*; when a communication channel is not empty, then we say that the user is *engaged in a call* (but not necessarily connected to another user).

The basic protocol for call setup from A to B is as follows, assuming neither are engaged in a call. When A goes off hook, the message (A,0) is placed on channel A. After dialling B, the message (A,0) is sent to channel B. When B receives this message, the message (B,1) is sent to channel A and the status bit in channel B is changed to 1; the connection is then established. At this point, the originating party is in state *oconnected* and the terminating party in state *tconnected*.

To clear down, A can close down one side of the connection by going on hook: the message is removed from its communication channel and the status bit of the message in channel B becomes 0. Then, neither A nor B are in a connected state, and A is free to close down the connection. On the other hand, channel B cannot close down the connection (reflecting the real-life situation whereby termination is asymmetric). So, if B goes on hook, while A and B are connected, then the connection status remains unchanged for both A and B.

Implementing the basic protocol and state transitions are reasonably s-traightforward, the main difficulty is with call cancellation, or "non-standard" behaviour during call setup. In an earlier model [2], several race conditions were uncovered through judicious use of assertions and assertion-checking. These are overcome here by extending messages to include both a channel and a status bit.

3.1 The Basic Call Service in Promela

Promela is a state-based formalism, rather than event-based. Therefore we represent events and device states by explicit variables, *call* states (e.g. calling, dialling, etc.) by labels and atomic statements, and transitions between states are represented by goto's. We note that confusion between high level *call* states (e.g. calling, dialling, etc.) and global or local states of the state-space underlying the Promela code is difficult to avoid. We will try to distinguish the former by referring explicity to them as *call* states, or by their specific names such as the *idle* state.

An example of the Promela code associated with the *idle* state and its outgoing transitions is given below. In section 4.3 we discuss improvements made to the model. The improved version of the *idle* state can be seen, along with the entire code (after features have been added) in the appendix. The global/local variables and parameters include the self-explanatory `selfid`, the communication channels `self` and `partner`, the variables `dev`, `event` and `network_event`. `messchan` and `messbit` are local variables used for reading messages. The channel `null` allows a default value for the `partner` variable when that call process is not engaged in a call. This value is not strictly necessary for modelling purposes, but can be valuable for reasoning.

```
proctype User(byte selfid; chan self)
...
idle:atomic{assert(dev == on);
            assert(partner[selfid]==null);
     /* either attempt a call, or receive one */
     if
     :: empty(self)->
        event[selfid]=off;
        dev=off;
        self!self,0;  goto dial
        /* no connection is being attempted, go offhook */
        /* and become originating party */

     :: full(self)-> self?<partner[selfid],messbit>;
        /* an incoming call, become terminating party */
        partner[selfid]?<messchan,messbit>;
        if
        :: messchan == self -> goto talert
          /* call attempt still there */
        :: else -> self?messchan,messbit;
                   partner[selfid]=null; goto idle
          /* call attempt cancelled */
        fi
     fi};
```

The service is defined a parameterised process; a network of users of that service is represented by the appropriate concurrent instantiations (i.e. `run user(...); run user(...);` etc.).

Since each *call* state transition is represented by an atomic statement, we have to ensure that no variable about which we intend to reason is updated more than once within each such statement. Hence the need to differentiate between *user-initiated* events (`event`) and *network-initiated* events (`network_event`). We note also that *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.

4 Analysis of Behaviour

Analysis concerns two issues. First, the asynchronous nature of the communication between call processes and the intricacies of call setup and clear down give rise to plenty of scope for errors in the underlying protocol. In particular, it can be difficult to handle call cancellation at an advanced stage of call setup correctly; it is quite easy to make incorrect assumptions about the state of a communication channel or to overlook an unexpected race condition, as discussed earlier. Second, feature interaction analysis involves checking that properties enjoyed by a feature (in any configuration) are still preserved in any other configuration (e.g. in the presence of other features). While in-line assertions and generic tests for livelock and deadlock are an important part of the checking process, here we concentrate on reasoning about domain specific temporal properties, expressed in LTL (linear temporal logic).

Expressing the behavioural properties required for feature interaction analysis in LTL is challenging, due to the distributed and asynchronous nature of the system. The issue is not just one of path quantifiers, i.e. it cannot be solved simply by moving to a more expressive logic such as CTL*; it is more subtle.

Standard LTL formulae are constructed from a set of atomic propositions, the standard Boolean operators ($\neg$, $\wedge$ and $\vee$), and the temporal operators $\Box$ (always), $\Diamond$ (eventually), $\circ$ (next) and U ((strong) until). Propositions include equivalences and process control such as $p@label$ meaning process p is at label $label$. The types of properties that we wish to verify consist of LTL formulas together with one of the two path quantifiers A meaning *for all paths* and E meaning *for some paths*.

When using the partial order reduction algorithm (see section 4.3) to verify properties using Spin, the temporal operator $\circ$ (next) is to be avoided, due to problems of *stuttering* equivalence. In fact, in our experience it is very rare that a property holds in every *next* global state in a distributed system. However, a much more useful concept would be a *relativised* $\circ$, that is a *next* state relative to a particular constituent process. This would allow us to formalise properties such as after a process p reads from a channel, its next action is to increment a variable in a certain way. Consequently, we introduce the operator $\circ_p$ meaning the next global state in which process p has made a (local) transition (i.e. the first global state in which p has made a move). Thus in general, $\circ$ and $\circ_p$ refer to different states, the former occurring before (or at the same time as) the latter. Clearly, the problems associated with stuttering equivalence still apply, and so partial order reduction can not be used when verifying properties involving this operator.

In order to allow a more compact representation of properties, we adopt some notation of [3]. That is we introduce the operators $\mathcal{W}$ (*weak* until) and $\mathcal{P}$ (precedes), defined as follows:

$$f\mathcal{W}g = \Box f \vee (fUg)$$

and

$$f\mathcal{P}g = \neg(\neg fUg).$$

4.1 Temporal Properties

The properties below are only a representative set; they are chosen to illustrate the diversity required for analysis and the role of relativised operators.

Further (global) variables are introduced for reasoning (and commented out when they are not used), these include the arrays *connect.to*, recording the presence of a connection between two users, *dialled*, recording the most recent number dialled (since leaving the *idle* state), and the *event* and *network_event* arrays that were mentioned earlier. The global *dev* array is replaced by the local *dev* variable when it is not needed for reasoning.

We note that we have taken property 3 from [10], to illustrate comparison with their results. Here *proci* and *chan_name*$[i]$ are (respectively) the process instantiation number and the channel name associated with user i. We assume that we are referring to a 4 user call process and that each of the variables i, j and k belong to the set $\{0, 1, 2, 3\}$.

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

Property 3 *Busy tone or ringing tone will always directly follow "trying".*
That is: $A\Box(p \rightarrow \circ_{proci}q)$ 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: $A\Box(p \rightarrow q)$ 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: $A\Box((p \wedge v \wedge t) \rightarrow (((\neg s)\mathcal{W}(w))||((\neg r)\mathcal{W}q)))$ where p is $dialled[i] == j$, v is $event[i] == dial$, t is $full(chan\_name[j])$, s is $event[i] == call$, w is $len(chan\_name[i]) == 0$ and r is $user[proci]@idle$ and$q$ is $network\_event[i] == engaged$, for $i \neq j$.

Notice 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 *Dialling immediately precedes reaching the calling state.*
That is: $A\ \Box(p \rightarrow q)$ where p is $user[proci]@calling$ and q is $event[i] == dial$.

We expect all properties to be *true*, for the basic service. Note that property 1 would not hold for *all* sequences because a connection may not always be possible, for example, because the other line is out of service, or constantly engaged, or the originator goes on-hook before a connection is made.

4.2 Proving Temporal Properties in Spin

SPIN enables us to check the satisfaction of an LTL formula over all execution paths. The mechanism for doing this is via *never* claims – processes which describe *undesirable* behaviour. Full details of never claims and Büchi automata are given in [7]. Here, we give a brief overview of the mechanisms involved and a description of how they have been employed.

For a given LTL formula f, a never claim has the form: *f is true for all execution paths*. From this never claim, a Büchi automaton is constructed which is a process which is run in lock-step with the system under investigation. If the never claim does not hold, the Büchi automaton will "catch" at least one execution sequence for which $\neg f$ is true. In this case the never claim either completes or reaches an *acceptance* cycle, and the never claim is said to be *matched*. It is also possible, when using XSpin (the X interface for Spin) to check whether a given property is true for no execution paths (by choosing the *No Executions* option rather than the *All Executions* option). That is, Spin offers automatic negation of the temporal logic property. Thus the properties Af and $A\neg f$ (where f is an LTL formula) are proved by checking that f is true for all executions and for no executions respectively. For example, to prove Property 1, of the form: "$\Diamond p$ holds for some path", one needs only show that the never claim "$\Diamond p$ holds for no execution paths" is matched. However, to prove all of the other properties, it is necessary to search the entire state-space.

Xspin provides a facility for translating standard LTL into never claims. However, to ensure that the never-claims generated in this way are stutter-closed, this translator does not allow the use of the next operator $\circ$ or, naturally, the relativised next operator $\circ_p$. In general, we use the (SI)-EQLTL conversion tool of Etissami [4] to translate LTL formulas into never-claims. This allows the use of the next operator. However, property 3, which includes the relativised next operator must be translated into standard LTL (see below).

4.2.1 Relativised Operators

Relativised operators can be implemented by judicious use of the built-in global variable *_last*. This variable holds the value of the (internal) process number of the process that last made a transition. The internal process number is assigned at process initialisation and can therefore be captured by an assignment at that time (e.g. `p0 = run user(...)`).

Thus Property 3 becomes

$$(p- > X(([](r))||((r)U(r\&\&q))))$$

where p is $event[i] == call)$, q is $(network_event[i] == engaged)||(network_event[i] == oring)$ and r is $\_last == proci)$.

Although some formulas that involve the next operator $\circ$ can be shown to be stutter-closed (and so partial order reduction can be used for the verification of such properties), the use of the *_last* variable precludes the use of partial order reduction.

4.3 Verification and Optimisation

Verification is performed on a network of user processes and the *number* of such processes is significant: too few and interactions or race conditions will not exhibit, too many and the state-space will prove intractable. For the basic service at least three processes must be considered, when features are included, this minimum can rise to four.

Initial reasoning with a network of three users proved successful, but with four users the search depth rose to in excess of 10^8– intractable on most machines. It seemed that our model was simply pathologically "bad" for model checking, particularly because the state-space was excessively deep (i.e. depth first search offers no gain).

This problem has also been encountered by others, notably by Plath and Ryan in [10], though because they have only synchronous communication and explicitly schedule the services, we would expect their state-spaces to be significantly smaller. A partial solution is to reduce functionality, for example to make one service originating behaviour only and another terminating behaviour only (these can be regarded as features). However, this is in general unsatisfactory; the question of why our state-space was so large, and what we could do about it vexed us for some time. *Is it possible to improve the efficiency of the state-space exploration, or transform the Promela model to generate a smaller space, without fundamentally altering the level of abstraction of the model?* Several possible approaches have been explored and are discussed below.

Partial order reduction

Partial order reduction (POR) (see [9, 6]) reduces the size of the state-space by exploiting the fact that many checked properties are insensitive to the interleaving of concurrent activities: certain permutations of "safe" transitions can be excluded. However, POR has only limited effect on our model. Closer examination shows that this is hardly surprising, since safe statements (transitions) are essentially assignments to local variables or exclusive channel read/send operations. The former are not only rare, but they are embedded in atomic statements that are themselves only safe if all component statements are safe. The latter do not appear at all. But why not?

Exclusive read/send channels

A channel is said to be *exclusive read-access* within a process, annotated xr, if no other process can read from the channel. Similarily, a channel is said to be *exclusive send-access* and annotated xs. In the model, there are a few channel instances which could be declared to be the latter, but none the former. Moreover, while we could declare further dedicated channels between pairs of processes, and annotate them appropriately, we are still left with the problem that even a non-destructive read or test of the length of a channel violates the xr property. Such a test is crucial: often behaviour depends on the exact contents of a channel. Thus, while some small gains can be made, they are minimal. Moreover, many such statements are embedded in unsafe atomic statements; it would clearly be a retrograde step to reduce the atomicity.

State compression

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 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, for example, after 65 hours, the depth reached was only of the order of 10^6.

Optimised never claims

The structure of never claims (Büchi automata) can affect efficiency and it is worthwhile noting that improvements in the generation of never claims from LTL formulae, in the current version of Spin (Version 3.3.10), that were inspired by the work of Etessami, [4], have led to the faster detection of "bad" paths for some properties. For example, the never-claim corresponding to Property 1 is:

```
never {    /* (<>  p) */
T0_init:
        if
        :: ((p)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

This differs from the claim generated in previous versions of Spin in that the transition out of state T0_init to the accept state (accept_all) occurs first within the if ... fi structure (rather than last, as in previous versions). This leads to a faster violation of the never-claim, since the depth-first search takes the first enabled move that has not yet been chosen. In the previous version the transition out of the T0_init state would not have been chosen until the system processes had completed a cycle.

Variable re-initialisation

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 re-initialised 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 re-initialised before subsequent visits. The largest reduction is to be found when variables are set before progressing to the next *call* state. Unfortunately, this is not always possible: as we have discussed earlier, variables about which we may intend to reason may not be updated more than once within each such atomic statement. However, there is a solution: add a further state where variables are re-initialised. For example, we have added a new state *preidle*, where the variables *network_event* and *event* are re-initialised, before progression to *idle*.

Note that although the (default) data-flow optimization option available with Spin attempts to reinitialize variables automatically, we have found that this option actually *increases* the size of state-space of our model. This is due to the initial values of variables being non-zero (when they are of type mtype for example). Therefore we prefer to switch this option off, and reinitialise our variables manually.

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.

Update of Specific Global Variables

By merely commenting in/out any reference to (update of) the *event* variables 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 update the value of $event[i]$ to the value *eventq*, and leave the other event variables set to their default value. 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.)

4.4 Verification after Optimisation

It was possible to verify all six properties fairly quickly and well within our 1.5 Gbyte memory limit. State compression was used throughout.

For the proof of property 1, the expected never-claim occurred within a maximum search-depth of $430,000$ in each case.

For each of the properties $2, 4, 5$ and 6 a search of the entire search-space showed there to be no errors. When partial-order reduction was applied each search was completed within a maximum search-depth of 2.4 million and there were at most 7 million stored states in each case. Each search completed within 45 minutes. Failure to apply partial-order reduction resulted in an increase in the maximum search depth reached of between 20% and 25% and a corresponding increase in the number of stored states of between 28% and 40%.

The verification of property 3 took longer (268 minutes), required a greater search-depth to be reached (5 million) and more states to be explored (2.7×10^8). This is 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.

5 Features

Now that the state space is tractable, a number of features are added to the basic service:

- **CFA – call forward always** 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.

Details of the features are not given here (the entire code for the User process is given in the Appendix, although the initialization process is not included). However we make a few observations:

- To implement the forwarding capabilites we have included an "in-line" translator function that computes the transitive closure of the forwarding relations (modulo call states, as forwarding may not be universal).
- 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.
- We restrict the size of the lists of screened callers (relating to the OCS, ODS and TCS features) to one. This is sufficient to demonstrate feature interactions, and limits the size of the state-space.

- The addition of ACB, while straightforward, increases the complexity of the underlying state-space greatly. This is because 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 (c.f. discussion above about variable re-initialisation).

5.1 Temporal Properties for Features

The construction of these properties was much more complex. In order to accurately reflect the behaviour of each feature great attention must be paid to the *scope* of each property. 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.

Property 7 – **CFA** 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: $A\Box(p \rightarrow (r\mathcal{P}q))$, 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: $A\Box(((u \wedge t) \wedge ((u \wedge t)U((!u) \wedge t \wedge p))) - > (!((!r)Uq)))$, where p is $dialled[i] == j$, t is $full(chan\_name[j])$, r is $partner[i] == chan\_name[k]$, u is $User[p0]@diall$ 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: $A\Box(\neg p)$, 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: $A\Box(\neg p)$, where p is $dialled[i] == 1$.

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: $A\Box(\neg p)$, 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 $event[i] == call$.

6 Analysis of Feature Behaviour

Features are considered individually, and then pairwise for interactions. For all properties we assume implicit quantification over all i,j; for properties 8–12, the constraints on i,j are given explicitly.

6.1 Property Validation Results

Each feature is validated against the appropriate set of properties (1–12). A $-$ denotes that the property is not appropriate– for example a property might depend upon data that is not relevant to that feature.

	BC	CFA	CFB	OCS	ODS	TCS	ACB
property 1	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$	$\surd$
property 2	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$
property 3	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$
property 4	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$
property 5	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$	$\surd$
property 6	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
property 7	$\times$	$\surd$	-	-	-	-	-
property 8	$\times$	-	$\surd$	-	-	-	-
property 9	$\times$	-	-	$\surd$	-	-	-
property 10	$\times$	-	-	-	$\surd$	-	-
property 11	$\times$	-	-	-	-	$\surd$	-
property 12	$\times$	-	-	-	-	-	$\surd$

6.2 Interaction Results

For interaction analysis, *pairs* of features (either pertaining to the same user or to two different users) are "switched on" that is, two features are assumed to be active in a suitable configuration.

For two features f_i and f_j, let $f_i \cup f_j$ denote a system with both features active. We say that two features f_i and f_j *interact* if for a property ϕ, we have $f_i \models \phi$ but $f_i \cup f_j \not\models \phi$. Namely, a property that holds for f_i alone, no longer holds in the presence of another feature f_j.

If we adopt a naive approach to interaction and consider any property as a candidate for ϕ, then it is easy to see that all features interact (since for each pair there is at least one property in the table above with a different result). Therefore, a more selective approach is required: we consider only the properties associated with the features under examination, namely for features f_i and f_j, consider properties ϕ_i and ϕ_j.

The following table gives the interactions found for pairs of features and the relevant properties from (7–12). A $\surd$ indicates that there exists some property for which an interaction is shown to exist for the pair of features, whereas a $\times$ indicates that no such interaction has been found. We have excluded BC as every feature interacts with it in some way.

	CFA	CFB	OCS	ODS	TCS	RBWF
CFA	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\surd$
CFB	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\surd$
OCS	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$
ODS	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
TCS	$\times$	$\times$	$\times$	$\times$	$\times$	$\surd$
RBWF	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\times$

It is important to note that the interactions uncovered depends very much on the *properties* chosen and how the features are *modelled*. The properties are not complete descriptions, in particular they do not state what should *not* happen (i.e. the frame problem). For example, this means that no interaction was found between CFA and CFB, when both features *pertained to the same user* when one might be expected. This is because property 8 describes only how CFB works when the dialled party is engaged, it does not state that that otherwise call behaviour is as for the basic service. One might also expect an interaction between TCS and CFA, but this was not found because although TCS will block the call, the *partner* variable will be set appropriately, thus satisfying property 7.

7 Conclusions and Future Work

We have presented a telecomms service model in Promela that permits truly independent call control processes with asynchronous communication, asymmetric call control, and a number of features. For the basic service, and each of the features, we have presented a number of domain-specific properties and their representations in the logic LTL. An important new aspect to LTL is a relativised next operator, allowing us to reason about the behaviour of constituent processes. This is valuable additional functionality, in this domain.

Initial attempts to reason about these properties, for systems composed of more than three service users, proved to be intractable. The standard, generic approaches to optimisation (e.g. partial order reduction, compression) made a minimal impact. However, we were able to transform the underlying Promela description, without abstracting away from the behaviour, to one with a significantly smaller state-space (about 10 percent of original). The simple "trick" is re-initialise Promela variables to ensure that there is a much closer relationship (i.e. approaching 1:1) between abstract call states and the underlying Promela states.

With the basic service stable, we added six further features. For each of these features, we also gave an associated property and LTL representation. These properties were used first, to validate the features (i.e. correctly implemented in the services) and then second, for feature interaction analysis. While this small set of properties does not provide a complete approach to feature interaction analysis, it did allow us to uncover most of the "usual suspects" and more importantly, demonstrates the feasibility of the model-checking approach.

In conclusion, our results represent a contribution to providing a fully verifiable model of a non-trivial service-level view of a telecommunications system, a domain where full verification is rare. We have tried to remain quite close to an operational view of the system, therefore our system has a high degree of asynchronicity and there is a tension between the desired operational view and state-space explosion. A major breakthrough is the way in which we have been able to control the state-space explosion. We believe that not only are our results interesting in themselves, but our experience provides useful guidance for others model-checking in this, and other, application domains.

Future work includes investigating generalising our results to systems composed of an arbitrary number of processes (i.e. induction), further relativised operators and symmetry based reduction techniques.

Acknowledgements

The authors would like to thank Gerard Holzmann for his help and advice, and the Revelation project at Glasgow for providing computing resources.

References

[1] M. Calder, E. Magill and D. Marples. A Hybrid Approach to Software Interworking Problems: Managing Interactions between Legacy and Evolving Telecommunications Services. *IEE Proceedings - Software*, Vol. 146, No. 3, June 1999.

[2] M. Calder and A. Miller. Analysing a Basic Call protocol using PROMELA/XSPIN. In Proceedings *SPIN '98: Workshop on Automata Theoretic Verification with the SPIN model Checker*, pp. 169-181, Paris, France, 1998.

[3] Marsha Chechik and Dimitrie O. Paun. Events in Property Patterns. In D. Dams, R. Gerth, S. Leue, and M. Massink (eds.),5th International Spin Workshop, Trento, Italy Theoretical and Practical Aspects of SPIN Model Checking, *Lecture Notes in Computer Science*, vol. 1680, pp. 154-167, 1999.

[4] K. Etessami. Stutter-invariant Languages, ω-automata, and temporal logic. In *CAV'99: Computer Aided Verification, 11th International Conference*, pp. 236-248, 1999.

[5] *IN Distributed Functional Plane Architecture*. Recommendation Q.1204, ITU-T, March, 1992.

[6] P. Godefroid. Partial Order Methods for the Verification of Concurrent Systems. *Lecture Notes in Computing Science*, vol. 1032, 1996.

[7] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering,* **23**, No.5, May 1997.

[8] Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. *Proceedings of the 1999 International Conference on Software Engineering*, pp. 597-605, 1999.

[9] Doron Peled. Combining Partial Order Reductions with On-the-fly Model Checking. *Formal Methods in System Design,* **8**, pp. 39-64, 1996.

[10] M. Plath and M. Ryan. Plug and Play Features. In C. Kimbler and L.G. Bouma, (eds.) *Feature Interactions in Telecommunications and Software Systems V.* IOS Press, pp. 150–164,1998.

[11] M. Thomas. Modelling and Analysing User Views of Telecommunications Services. In P. Dini, R. Boutaba, and L. Logrippo, (eds.) *Feature Interactions in Telecommunications Systems IV.* Montreal, IOS Press, 1997.

Appendix: The Basic Service with Features

```
mtype = {on,off,dial,call,oring,tring,
         unobt,engaged,connected,
         disconnect,callback};
chan null = [1] of {chan,bit};

chan zero = [1] of {chan,bit};
chan one = [1] of {chan,bit};
chan two = [1] of {chan,bit};
chan three = [1] of {chan,bit};

chan chan_name[4];
/* convert from no. to chan name */

byte call_forward_always[4];
byte call_forward_busy[4];
byte orig_call_screen[4];
byte orig_dial_screen[4];
byte term_call_screen[4];
/* ith element of array set to 6*/
/*if user[i] doesnt have feature*/
/* and to id of user to be forwarded*/
/* to or screened from otherwise*/

byte ring_back_when_free[4];
/*set to 0 or 1*/
/*feature off or on*/
byte record[4] = 6;
/*mtype dev[4] = on;*/
byte dialled[4] = 6;
mtype network_event[4] = on;
mtype event[4] = on;
chan partner[4];

typedef array { byte to[4] }
    array connect[4];
   /* 16   bytes in  total */

short p0,p1,p2,p3=-1;

/*inline translator (q1,id1)

{
 do
 ::((call_forward_always[id1]!=6)||
    (call_forward_busy[id1]!=6
                    &&full(q1)))
   ->id1=(((call_forward_always[id1])
         +(call_forward_busy[id1]))%6);
         q1=chan_name[id1];
 ::((call_forward_always[id1]==6)&&
     (((call_forward_busy[id1]!=6)
        &&empty(q1))
   ||(call_forward_busy[id1]==6))
   )->break
 od
 }*/

inline event_action (eventq)

{
 if
 ::selfid==0->event[selfid]=eventq
 ::selfid!=0->skip
 fi
}

inline network_ev_action (neteventq)

{
 if
 ::selfid==0->
   network_event[selfid]=neteventq
 ::selfid!=0->skip
 fi
}
```

proctype User(byte selfid; chan self)

```
{/* start User */
chan messchan;
bit messbit;
mtype dev = on;
byte partnerid=6;
```

idle:

```
atomic
 {assert((dev== on)&&
       (partner[selfid]==null));
 /* either attempt call, or receive one */
  if
  :: empty(self)->event_action(off);
     dev=off;
     self!self,0;goto diall
     /* no connection being attempted*/
     /*go offhook */
     /* and become originating party */
  :: full(self)->/* an incoming call */
     self?<partner[selfid],messbit>;
     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 preidle
     fi
   fi};
```

diall:

```
atomic
 {assert((dev == off)&&full(self));
  assert(partner[selfid]==null);
  /* dial or go onhook */
  if
  :: event_action(dial);
    /* dial and then nondeterministic*/
    /*choice of called party */
     if
     ::orig_dial_screen[selfid]!=0->
       partner[selfid] = zero;
            dialled[selfid] = 0;
            partnerid=0
     ::orig_dial_screen[selfid]!=1->
            partner[selfid] = one ;
            dialled[selfid] = 1;
            partnerid=1
```

```
   ::orig_dial_screen[selfid]!=2->
     partner[selfid] = two ;
             dialled[selfid] = 2;
             partnerid=2
   ::orig_dial_screen[selfid]!=3->
             partner[selfid] = three;
     dialled[selfid] = 3;
             partnerid=3
   ::ring_back_when_free[selfid]==1->
     partnerid= 7;goto ringback
   fi;
   translator(partner[selfid],partnerid);
   if
   ::orig_call_screen[selfid]==partnerid
     -> goto unobtainable
   :: else->goto calling
   fi
::event_action(on); dev=on;
  self?messchan,messbit;
  assert(messchan==self); messchan=null;
  messbit=0; goto preidle
  /*go onhook, without dialling */
fi};
```

calling:

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

busy:

```
/* number called is engaged*/
/*go onhook*/
atomic
{assert(full(self));
 network_ev_action(engaged);
 event_action (on);dev = on;
 self?messchan,messbit;
 assert(messchan==self);
 partner[selfid]=null;
 partnerid=6;messchan=null;
 dialled[selfid]=6;
 messbit=0; goto preidle
 /*go onhook*/
 /*cancel connection attempt */
};
```

ringback:

```
/*comment out entire state when*/
/*ring-back_when_free not used*/
atomic
{self?<messchan,messbit>;
 assert (messchan==self);
 if
 :: self?messchan,messbit;dev=on;
    dialled[selfid]=6;
    partner[selfid]=null;partnerid=6;
    messchan=null;messbit=0;
    event_action(on);goto preidle
 :: (record[selfid] !=6) ->
    messchan=null;messbit=0;
    partner[selfid]=
    chan_name[record[selfid]];
    partnerid=record[selfid];
    event_action(callback);
    goto calling
 fi};
```

unobtainable:

```
/* number called unobtainable*/
/*go onhook*/
atomic
{assert(full(self));
 network_ev_action(unobt);
 event_action(on);dev = on;
 self?messchan,messbit;
 assert(messchan==self);
 partner[selfid]=null;partnerid=6;
 dialled[selfid]=6;messchan=null;
 messbit=0;goto preidle
 /*go onhook*/
 /* cancel connection attempt */
 };
```

oalert:

```
/* called party is ringing */
atomic
{assert(full(partner[selfid]));
 assert(full(self)&&(dev == off));
         network_ev_action(oring);
 self?<messchan,messbit>;
 assert(messchan==partner[selfid]);
 messchan=null;
 /* check channel */
 if
 ::messbit== 1->messbit=0;
   goto oconnected
  /* correct token */
 ::messbit==0->goto oalert
 /* wrong token*/
 /*not connected yet, try again */
 ::messbit==0->goto oringout
   /* give up */
 fi};
```

oringout:

```
/*abandon call attempt*/
atomic
{assert(full(partner[selfid]));
 assert(full(self)&&dev==off);
 event_action(on);dev=on;
 self?messchan,messbit;
 partner[selfid]?messchan,messbit;
 partner[selfid]!messchan,0;
 partner[selfid]=null;partnerid=6;
 dialled[selfid]=6;
 messchan=null;messbit=0;goto preidle;
 /* give up, go onhook */
};
```

oconnected:

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

oclose:

```
/* disconnect call */
atomic
{assert(full(self));
 assert(full(partner[selfid]));
 event_action(on);dev = on;
 self?messchan,messbit;
 /* empty own channel */
 assert(messchan== partner[selfid]);
 assert(messbit==1);
 partner[selfid]?messchan,messbit;
 /* empty partner's channel*/
 assert(messchan==self);
 assert(messbit==1);
 /* and disconnect partner */
 partner[selfid]!messchan,0;
 connect[selfid].to[partnerid] = 0 ;
 partner[selfid]=null;
 dialled[selfid]=6;
 partnerid=6;messchan=null;
 messbit=0;goto preidle};
```

talert:

```
atomic
{assert((dev == on)&&(full(self)));
 /* either device rings or*/
 /* connection attempt is cancelled*/
 /* and then empty channel */
 partner[selfid]?<messchan,messbit>;
 if
 :: messchan==self->
    network_ev_action(tring);
    messchan=null;messbit=0;
    goto tpickup
 :: else->
    /* attempt has been cancelled */
    network_ev_action(disconnect);
    self?messchan,messbit;
    partner[selfid]=null;partnerid=6;
    dialled[selfid]=6;
    messchan=null;messbit=0;
    goto preidle
 fi;
};
```

tpickup:

```
/* proceed with connection*/
/* or connect attempt cancelled */
atomic
{assert(full(self));
 if
 :: full(partner[selfid]) ->
    partner[selfid]?<messchan,messbit>;
    if
    ::messchan==self ->
    /*connection proceeding */
      assert(messbit ==0);
      self?messchan,messbit;
      assert(messchan==partner[selfid]);
      assert(messbit==0);
      event_action(off);dev = off;
      partner[selfid]?messchan,messbit;
      partner[selfid]!self,1;
      /* establish connection */
      self!partner[selfid],1;
      messchan=null;messbit=0;goto tclose
    ::else ->/* wrong message*/
      /*connection cancelled */
      network_ev_action(disconnect);
      self?messchan,messbit;
      event_action(on);dev=on;
      partner[selfid]=null;
      dialled[selfid]=6;
      partnerid=6;messchan=null;
      messbit=0;goto preidle
    fi
 ::empty(partner[selfid])->
   /* connection cancelled */
   network_ev_action(disconnect);
   self?messchan,messbit;
   event_action(on);dev=on;
   partner[selfid]=null;
   partnerid=6;dialled[selfid]=6;
   messchan=null;messbit=0;
   goto preidle
 fi};
```

tclose:

```
/* check if originator*/
/* has terminated call */
atomic
{self?<messchan,messbit>;
 if
 ::(messbit == 1 && dev == off)
    -> event_action(on);
   /* trivial handset down */
   dev = on; messchan=null;
   messbit=0;goto tclose
 ::(messbit == 1 && dev == on)
    -> event_action(off);
    /* trivial handset up */
   dev = off; messchan=null;
   messbit=0;goto tclose
 ::(messbit == 0 && dev == on)->
 /*connection is terminated*/
   self?messchan,messbit;
   partner[selfid]=null;partnerid=6;
   dialled[selfid]=6;messchan=null;
   messbit=0;goto preidle
 ::(messbit == 0 && dev == off) ->
   network_ev_action(disconnect);
   /* disconnect tone */
   event_action(on);dev=on;
 /*connection is terminated */
   self?messchan,messbit;
   partner[selfid]=null;dialled[selfid]=6;
   partnerid=6;messchan=null;
   messbit=0;goto preidle
 fi};
```

preidle:

```
atomic
{network_ev_action(on);
 event_action(on);
 goto idle
}
}/*end User */
```