Using the Model Checker SPIN to Detect Feature Interactions in Telecommunications Services

M. Calder and A. Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.

Abstract

The model checker SPIN is applied to the analysis of feature interactions in the design of telecommunications services. The application area is a challenging one for model-checking because formulating the right temporal properties in a distributed system is difficult and the state spaces quickly become intractable. We demonstrate how to express the properties in LTL and give minimal abstraction techniques that can greatly reduce the cost of model-checking. We also show how analysis can be performed automatically using scripts.

Key words: Promela/SPIN; communicating processes; distributed systems model checking feature interaction telecommunications services.

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, at various stages in the life-cycle, usually 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 services domain (for example, see [3]),

* Muffy Calder
Email address: muffy@dcs.gla.ac.uk (M. Calder and A. Miller).

though it exhibits in many other well-known domains such as email and electronic point of sales. We expect interactions to be an issue in next generation systems as well, for example in Grid technologies [21]. It is therefore important to have a range of techniques for dealing with them.

Techniques to deal with feature interactions can be characterised as design time or run time, interaction detection and/or resolution. Here, we concentrate on the detection of interactions at design time, with resolution through redesign.

When there is a proliferation of features, as in telecommunications services, then automated detection techniques are essential. In this paper, we investigate the feasibility of our specification and modelling approach, and the use of model-checking techniques, for feature interaction detection in *POTS* (plain old telephony) services. Much is known about the POTS domain, we therefore believe it to be an ideal domain to test the feasibility of our approach. We choose the model-checker SPIN [27], because of the suitability of the associated high level description language, Promela, for specifying software systems. Preliminary results for a system involving no features were reported in [6].

Model-checking involves constructing a finite model of a system and checking that a desired property holds in that model by exploring the state-space of the model. Theorem proving, on the other hand, involves deriving theorems from a given set of axioms. The former is ideally suited to our domain because telecommunications services are inherently concurrent and each model (essentially a labelled transition system) can be generated automatically from a high level description of a service. Consequently, the high level Promela descriptions can be modified with very little cost, seldom the case in a theorem proving approach where theorems have to be reproved, as the underlying theory changes.

Our approach involves considering a given service (and features) at two different levels of abstraction: communicating finite state automata and temporal logic formulae, represented by Promela specifications, labelled transition systems and Büchi automata. We make contributions in several ways, 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,
- state-space reduction techniques for Promela which result in tractable state-spaces, thus overcoming classic state-explosion problems,
- a technique for implementing a *relativised* temporal operator, that is one which depends on constituent processes, in the linear temporal logic of SPIN,
- 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.

Additionally, our results may serve to provide useful guidance for model-checking in this, and other, application domains.

The paper is organised as follows. Our overall approach to interaction detection, and the role of SPIN, is given in section 2. Following this, we give a short overview of telecommunications services, the language Promela, and then in some detail, how SPIN's search and verification algorithms work.

Sections 4 and 5 give an overview of the finite state automata and temporal properties for the basic service. In section 6 we give the Promela implementation of the basic call service, and optimisations to reduce the state space; in Section 7 we validate the basic service. Sections 8, 9 and 10 give similar descriptions and implementation of the features. Static and dynamic interaction analysis, respectively, are introduced in Sections 11 and 12. In Section 13 we automate the (dynamic) analysis and model generation and give results. In Section 14 we discuss the role of static and dynamic analysis. Conclusions and directions for further work are given in section 15.

Some of the results contained in this paper have been presented in [5]. However here we include greater implementation detail and provide more background material relating to SPIN. In addition, our analysis concerns 8 features whereas in [5] only 6 were considered.

1.1 Related Work

Model-checking for feature interaction analysis has been investigated by others, for example, using SMV [39], Caesar [41], COSPAN [17] and SPIN in [29]. 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 [39], the authors are restricted to two subscribers of the service with full functionality (plus two users with half functionality), due to state-space 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 [17], features and the basic service are described at only one level of abstraction, by temporal descriptions. State-space explosion is avoided, but interactions arising from implementation detail, such as race conditions, cannot be detected. Our layered approach permits detection of interactions arising from implementation detail, building upon earlier work by the first author (of this paper) in [41], where process algebra was employed. This too suffered from limitations

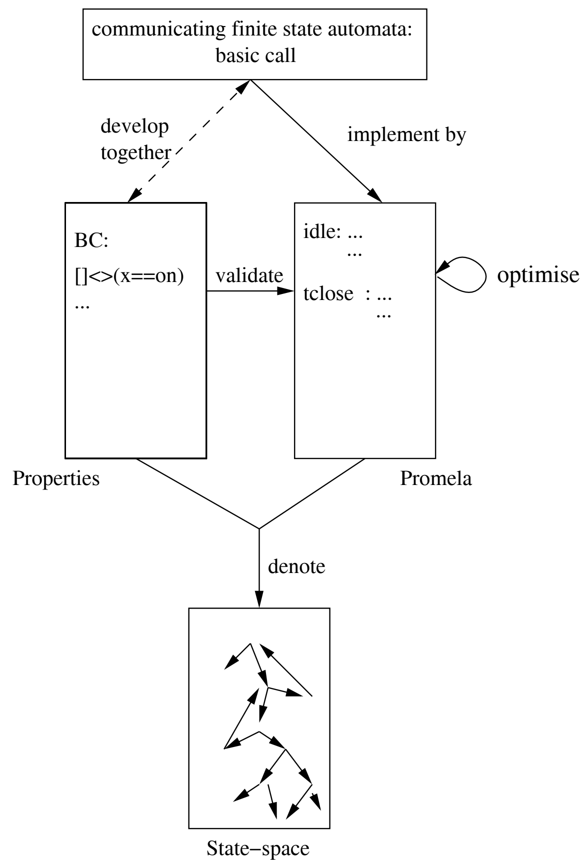

Fig. 1. Overall Approach - Basic Service

imposed by state-explosion and the lack of (explicit) asynchronous communication; indeed these limitations motivated the current investigation of Promela and SPIN.

2 Approach

Our approach has two phases; in the first phase we consider only the basic call service, as depicted in Figure 1. 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: 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 description in the middle of Figure 1 is regarded as the implementation; a crucial step is therefore validation of the implementation. This is done by checking satisfaction of the properties, using SPIN. Initial attempts fail, due to state-space explosion, however, an examination of the underlying state-

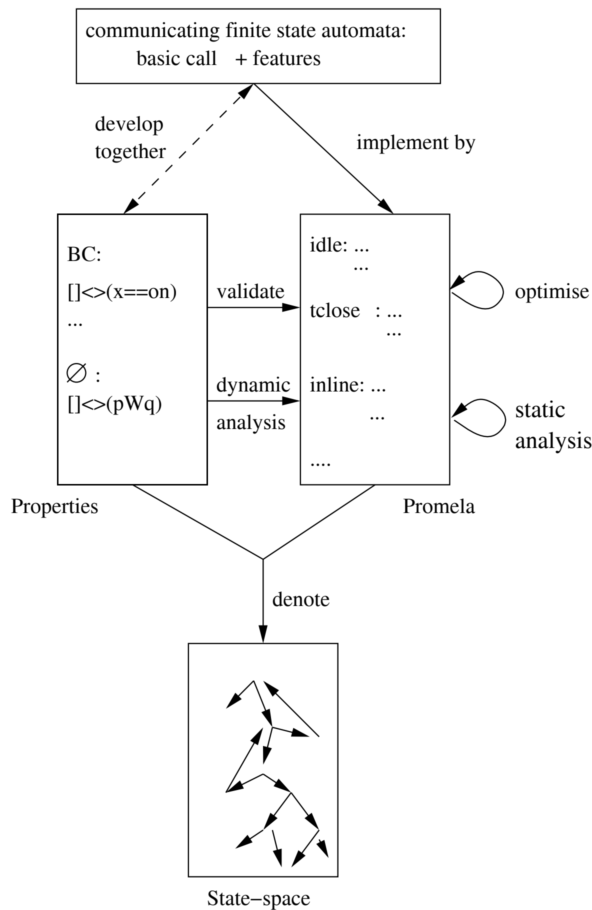

Fig. 2. Overall Approach - Basic Service + Features

space (far right of Figure 1) leads us to discover simple, but very effective optimisations.

The second phase, when we add features, is depicted in Figure 2. 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 "in-line" functions (see Section 10.1); again, the Promela has to be validated, this time against the feature properties. This leads us to our ultimate goal, interaction analysis, which takes two forms: *static* analysis, involving (syntactic) inspection of the Promela "code", and *dynamic* analysis, involving reasoning over combinations of sets of logical formulae and configurations of the feature subscribers, using SPIN. Both of these forms of analysis are indicated in Figure 2. The results of (either) analysis is interaction *detection*, often with a clear indication of possible resolution(s). The relationship between the two types of analysis is discussed in Section 14.

3 Background

3.1 Telecommunications Services

Control of the progress of calls is provided by a service at an exchange (a *stored program control* exchange). The service responds 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 will happen when there is a further incoming call? (Full details of all features mentioned here are given in section 8.) 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) [7] 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. For example, consider the scenario where 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 feature CFB, then A's feature OCS is compromised. Clearly, if the call is not forwarded, then the CFB feature is compromised. These kind of interactions can be very difficult to detect (and resolve), particularly since different features may be activated at different stages of a the call cycle.

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 [4]).

As with all distributed systems, there are many perspectives of a telecomms network, with varying levels of abstraction. Here, we have chosen to examine the user perspective of the *basic call service*, following the IN (*Intelligent Networks*) model, distributed functional plane [30].

3.2 Promela and SPIN

This section is intended to give an overview of Promela and SPIN. A more detailed description of the search algorithms and parameters used by the latter is included for the interest of the non SPIN expert, and as such can be omitted by those readers already familiar with Promela and SPIN.

Promela, *Pro*cess *me*ta *la*nguage [26,27], is a high-level, state based, language for modelling communicating, concurrent processes. It is an imperative, C-like language with additional constructs for non-determinism, asynchronous and synchronous communication, dynamic process 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 and provides several reasoning mechanisms: assertion checking, acceptance/progress states and cycle detection and satisfaction of linear temporal logic (LTL) formulae.

Other high-quality model-checkers include SMV, Murphi and FDR [34,12,40]. We choose to work with SPIN primarily because of the rich expressive power of Promela, and its suitability for modelling software processes. We do not choose SMV, for example, because the modelling languages: extended SMV and synchronous Verilog, were designed with hardware in mind. These languages do not allow one to model software as naturally and do not include asynchronous communication or explicit concurrency as primitives.

3.2.1 On-the-fly Depth-first Search

In order to perform verification on a model, SPIN converts a Promela specification into a labelled transition system (LTS). An LTS is defined as a triple $F = (S, f_0, T)$ where S is a finite set of states, f_0 is a distinguished initial state in S, and T a finite set of transitions, that is a set of pairs (s_1, s_2), where $s_1, s_2 \in S$. Each transition of the LTS corresponds to the execution of a specific atomic statement within one of the (concurrent) processes. An LTS can be represented by a graph (a *state-graph*) in which the nodes correspond to the states in S and directed edges correspond to the transitions in T.

A basic depth-first search (to check for deadlock, assertion violations etc.) explores the state-graph associated with F, starting from the initial state f_0, successively progressing along the edges of the graph and back-tracking when a previously visited state is reached, until an error is found – or until the entire search space has been explored.

3.2.2 Partial Order Reduction

Partial order reduction (POR) (see [36] and [35]) is a technique that is used to diminish the time and memory requirements when model-checking concurrent processes. It is based on the observation that execution sequences (or "traces") of a concurrent program can be divided into equivalence classes whose members are indistinguishable with respect to a property that is to be checked. By ensuring that at least one trace from each equivalence class is executed during a reduced search, the use of POR ensures that redundant work is not performed and that the truth (or otherwise) of a property is preserved.

A traditional, exhaustive, depth-first search of the state-space of a concurrent system involves the exploration of all of the transitions enabled at any state encountered during the search. The crux of the POR algorithm (described in [35]) is the fact that, in some cases, it is sufficient to explore a subset of the enabled transitions of a state (and hence reduce the total number of execution paths to be explored). Such a subset is called an ample set, and only exists for some states – when certain conditions are satisfied.

The implementation of POR used in SPIN (see [28]), involves the identification of various categories of Promela statement that can be statically marked as "safe" (or "conditionally safe") transitions. A subset of transitions enabled from a given state can only be ample if it consists entirely of safe transitions (and no successor of the state arising from any of these transitions has already been encountered during the current execution). The statements (transitions) that are marked as safe are essentially assignments to local variables or exclusive channel read/send operation. A channel is said to be *exclusive read-access* within a process, annotated xr, if no other process can read from the channel. Similarly, a channel is said to be *exclusive send-access* and annotated xs.

3.2.3 Parameters and Further Options used in SPIN Verification

One important parameter that needs to be set for any verification run is the *maximum search depth* M. This value determines the maximum distance along any path (starting from the initial state) that the depth-first search will explore. If M is set to a value that is less than the length of the greatest path from the initial state in a full search, the search will be *truncated*. In this case, whenever the search reaches depth M, the error message "maximum search depth too small" is reported, and the search will back-track until it reaches a previously unvisited state as before. This means that any state that is not reachable from the initial state along a path of length at most M, will not be explored.

For debugging purposes, a truncated search is suitable. Once an error has been reported, successive runs should be performed to determine the smallest value

of M for which the error will be "caught". If the search is truncated, the error will occur at or near the maximum search-depth. Following a verification with this maximum search-depth with a guided simulation will provide a description of the shortest path to the error.

If, on the other hand, after strenuous debugging, there are no errors believed to be present in the model, a full search-space search must be performed to show that there are no errors *at any depth*. In this case a truncated search is not suitable and M must be set to a value greater than the length of the greatest execution sequence. However (especially when conservation of memory is crucial), it is important that care is taken to set the maximum depth to as small a size as possible – to avoid the provision of an unnecessarily large (current search state storage) search stack.

For every reachable state, the corresponding *state-vector* is a unique characterisation of the state consisting of a sequence of bits in memory. During a depth-first search these states are usually stored in a hash-table. The size of the hash-table is determined via the *Estimated State-Space Size* parameter. If this is set too high, the memory required for the hash-table will be too great and verification will not be possible. If however this value is set too low the hash-table will not be large enough to accommodate all of the reached states. In general, it is advisable to leave this parameter set to the default value unless the Minimised Automaton Encoding (combined with Compression) or Supertrace options are selected. *Compression* is a method by which each individual state is encoded in a more efficient way. The memory required for each individual stored state is thus reduced, although the memory required for the hash-table itself remains unchanged. If MA-encoding is used without COM a hash-table is not used, and so the value of the *Estimated State-Space Size* parameter is irrelevant. However, if MA-encoding and COM are combined, a (small) hash-table is still required for the compression of the individual states. It is therefore prudent to set the *Estimated State-Space Size* parameter to a low value. Since we did not employ *Supertrace*, we omit its description here.

Finally, the *Weak Fairness* option [19,35,1] ensures that any process that has a transition that remains enabled will eventually execute it. The algorithm is based on a variant of Choeka's flag algorithm [8] and involves the construction of an extended state-space consisting of N copies of the original state-space (where N is the number of processes). Clearly the additional memory and time requirements of such an algorithm are great and therefore its use should be avoided where possible (see section 5.3).

As well as enabling a search of the state-space to check for deadlock, assertion violations etc., SPIN allows the checking of the satisfaction of an LTL formula over all execution paths. The mechanism for doing this is via *never claims* – processes which describe *undesirable* behaviour, and Büchi automata – automata that accept a system execution if and only if that execution forces it to pass through one or more of its accepting states infinitely often. Full details of never claims and Büchi automata are given in [27,18,33]. Here, we give a brief overview of the mechanisms involved and a description of how they have been employed.

Standard LTL formulae are constructed from a set of atomic propositions, the standard Boolean operators ($\neg$, $\wedge$ and $\vee$), and the temporal operators $[]$ (always), $\langle\rangle$ (eventually), $\circ$ (next) and U ((strong) until). Propositions include process control such as $p@label$ meaning process p is at label $label$.

When SPIN is used to verify an LTL property one must first use SPIN's LTL converter (or an alternative– see section 6.1) which translates LTL formulae into Promela syntax. This translation is a *never-claim* and encodes the Büchi acceptance condition. During a SPIN verification, this never-claim is converted to a Büchi automaton. Another Büchi automaton, consisting of the the synchronous product of the LTS corresponding to the concurrent system (model) and the Büchi automaton corresponding to the never-claim, is constructed. A depth-first search explores the state-graph associated with this (new) Büchi automaton.

If the original LTL formula f does not hold, the depth-first search will "catch" at least one execution sequence for which $\neg f$ is true. If f has the form $[]p$, (that is f is a *safety* property), this sequence will contain an *acceptance state* at which $\neg p$ is true. In this case the never-claim is said to *complete*. Alternatively, If f has the form $\langle\rangle p$, (that is f is a *liveness* property), the sequence will contain a cycle which can be repeated infinitely often, throughout which $\neg p$ is true. In this case the never-claim is said to contain an *acceptance cycle*. In either case the never claim is said to be *matched*.

When using SPIN's LTL converter it is possible to check whether a given property holds for *All Executions* or for *No Executions*. A universal quantifier is implicit in the beginning of all LTL formulas and so, to check an LTL property it is natural, therefore, 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*. This is not possible using LTL alone. However, SPIN can be used to show that "p holds for *No Executions*" is **not** true (via a never-claim violation), which is equivalent. Therefore, when listing our

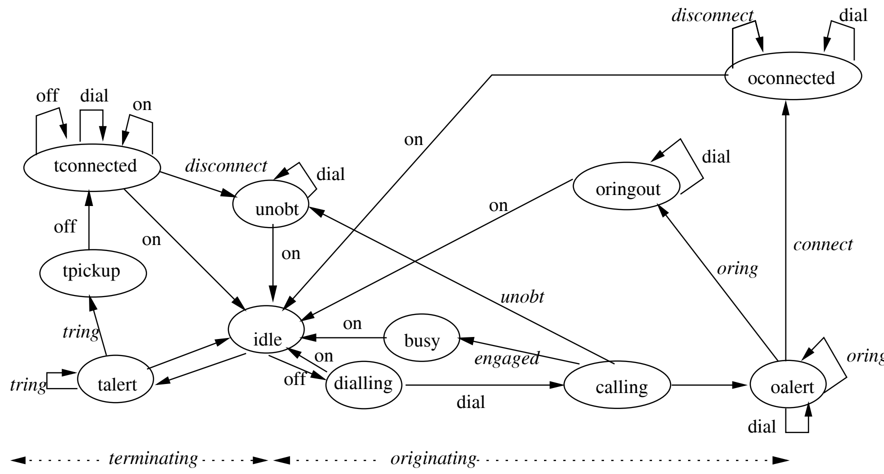

Fig. 3. Basic Call - States and Events

properties (section 5.3), we use the shorthand $E\langle\rangle p$ (meaning *for some path* p) to mean "($\langle\rangle p$ for No Executions) is not true".

This concludes the background material, we are now ready to begin the first phase of the approach: a description of the basic call service.

4 Basic Call Service

Figure 3 gives a diagrammatic representation of the automaton for the basic call service. 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. This reflects the fact that the ringing tone is indeed generated at each terminal device. Not all transitions are labelled. For example, there is an unlabelled transition from the (originating) state *calling* to *dial*, simply because there is no *observable* event associated with this transition.

The automata must communicate with each other; namely, the behaviour of one call process, as originating party, affects the behaviour of another call process, as terminating party. We do not adopt a formal notation to describe the communication mechanism (e.g. extended finite state automata) but describe it informally, as follows.

A communication channel is associated with each call process (or automata). 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

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 terminating party B is attempting connection
(B,1)	If channel B contains (A,1) then A and B are connected

Fig. 4. States of a Communications Channel in the Protocol

the connection). When it is not confusing, we refer to communication channel associated with call process A as channel A. When a communication channel is empty, then its associated call process is not connected to, or attempting to connect to, any other call process. When a communication channel is not empty, then the associated call process is *engaged in a call*, but not necessarily connected to another user. The interpretation of messages is described more comprehensively in Figure 4.

The communication channels are used to coordinate call set up and clear down. The basic protocol for call set up 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 the message on channel B is changed to 1; the connection is then established. 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). So, if B goes on hook, while A and B are connected, then the connection status remains unchanged for both A and B.

5 Basic Call Service Properties

In this section we give a set of temporal properties for the basic call service. Before doing so, we explain the form of the propositions and the addition of a new temporal operator.

5.1 Propositions

Propositions in SPIN's version of LTL may refer to values of (global) variables or to process "counters". Examples of the former are $x == 0$ and $x >= y$. An example of the latter is $user[proci]@idle$, meaning the incarnation of the process *user* with process identifier *proci* is at label *idle*. Process identifiers are simply global variables, initialised when a process is instantiated (and captured by assignment within the Promela *run* command).

In our propositions we will assume that the variables 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 *event* and *network_event*, for user-initiated and network-initiated events, respectively. *proci* and $chan_name[i]$ are the process identifier and the channel name associated with user process i, respectively.

5.2 Relativised Next Operator

When model-checking, the temporal operator $\circ$ (next) in LTL must be used with caution, due to problems of *stuttering* equivalence – properties involving $\circ$ can not be guaranteed to be closed under stuttering (see, for example, [37]). Consequently a valuable state reduction strategy, partial order reduction (see section 3.2.2) is not applicable in the presence of $\circ$. However, this is not a great hardship since it is in fact very rare that a property holds in every *next* global state in a distributed system.

The $\circ$ operator is of little use in a distributed system, what is required is a *relativised* $\circ$. That is, we would like to refer a *next* state *relative* to a particular constituent process. Such an operator would allow us to formalise properties such as "after a process p reads from a channel, *its* next action is to increment a given variable". (Note that the next global state might be one in which *another* process updates a *different* variable.)

We propose that in SPIN, relativised operators can be implemented by judicious use of the built-in global variable *_last* and the (LTL) *next* operator. This variable holds the value of the (internal) process number of the process that last made a transition. Although some formulae 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), in general, the use of the *_last* variable precludes the use of partial order reduction.

We introduce the operator $\circ_{proci}$ as a shorthand meaning the next global state in which process *proci* has made a (local) transition (i.e. the first global state

in which *proci* has made a move). In general, $\circ$ and $\circ_{proci}$ will refer to different states, the former occurring before (or at the same time as) the latter. Thus a property of the form "if p is true, then the next time process i makes a transition, q will be true", which is expressed in LTL as follows: $[](p \rightarrow \circ(([(]\neg(r))||((\neg r)U(r \wedge q))))$ (where r is defined as $_last == proci$), can be expressed more simply as $[](p \rightarrow \circ_{proci} q)$.

5.3 Basic Call Service Temporal Properties

As described in sections 3.3 and 5.2 respectively, we use the following shorthand notation: E (for some path) and $\circ_{proci}$ (next, with respect to process i). In order to allow a further compact representation of properties, we introduce the operators $\mathcal{W}$ (*weak* until) and $\mathcal{P}$ (precedes), defined as follows:

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

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). 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 $dialled[i] == chan_name[i]$, 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: $[]((p \wedge v \wedge t) \rightarrow (((\neg s)\mathcal{W}(w))||((\neg r)\mathcal{W}q)))$ 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 cannot 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 $event[i] == dial$.

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.

Great care has been taken to ensure that each temporal formula not only expresses precisely a property, but that the form will enable us to reason about them in the most efficient way. For example, it would be tempting to express Property 2 as $[](p \rightarrow \langle\rangle q)$, where p is $(dialled[i] == chan_name[i])$ and q is $(network_event[i] == engaged)$ (see [6]). This formula would be problematic in two ways. On the one hand it could be satisfied in a situation where a caller dialled his/her own number but failed to hear the engaged tone as a result (but heard the engaged tone *ultimately*, albeit during a different call). This would result in a longer search time to find an error. On the other hand, this formula would cause an error to be reported if a caller dialled his/her number and then simply *failed to progress* infinitely often. To avoid this unwanted scenario, the weak-fairness option would be required (see section 3.2.3), so causing a huge increase in the search-depth/time. The use of the $\mathcal{W}$ operator in this situation is therefore crucial.

6 Basic Call Service in Promela

Each call process (see figure 3) 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 Promela code associated with the *idle*, *dialing*, *calling* and *oconnected* states and their outgoing transitions is given below. The global/local variables and parameters include the self-explanatory `selfid` and `partnerid`, the communication channels `self` and `partner`, and the variables `dev`, `dialled`, `event` and `network_event`. In addition `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.

Any variable about which we may intend to reason should not be updated more than once within any atomic statement, other variables may of course be updated as required. In addition d_steps (deterministic sequences of code that are executed indivisibly), 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. Finally, we note that there are numerous in-line 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 dialing
     /* 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};


dialing:
  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)}
```

6.1 Options and State-space Reduction

Initial attempts to validate the properties against a network of four call processes fail because of state-space explosion. In this section we examine the causes of state-space explosion, the applicability of standard solutions involving configuring SPIN and how the the Promela code itself can be transformed to optimise the state-space. The fully optimised code (including features) is given in the appendix.

SPIN Options

The most obvious, standard optimisation to apply is POR (see section 3.2.2); however, it has only limited effect on our model. Closer examination shows that this is hardly surprising. The only statements statically defined as "safe" by SPIN are 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: there are a few channel instances which could be declared to be xs, 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.

States can be compressed using *minimised automaton encoding* (MA) or *compression* (COM) (see section 3.2.3). 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, for example, after 65 hours, the depth reached was only of the order of 10^6.

State-space Reductions

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 a variable (about which we wish to reason) being updated more than once within an atomic statement (as discussed in 6). 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.

By merely commenting in/out any reference to (update of) all 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 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, though 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.

Unlike other abstraction methods (see for example [9], [20] and [23]) our techniques are simple, and merely involve making simple checks that unnecessary states have not been unintentionally introduced. We believe that these kinds of state-space explosions are not uncommon. All SPIN users should be aware that they may be introducing spurious states when coding their problem in Promela.

Finally, we note that the *structure* of never claims (Büchi automata) can affect efficiency and consistency of results. We use the conversion tool of Etessami, [16], which will shortly be implemented in SPIN. The Büchi automata in this case tend to be smaller (that is they contain fewer states) and have led to the faster detection of "bad" paths for some properties.

7 Basic Call Service Validation

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

In property 1 the *No Executions* option was selected and for all other properties, the *All Executions* option was selected.

For the verification of property 1, a path containing the expected never-claim violation was found within a search-depth of $10,000$ 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 2.5 million and there are at most 1 million stored states in each case. Each search completed within 10 minutes. Failure to apply partial-order reduction resulted in an increase in the maximum search depth reached of between 19% and 24% and a corresponding increase in the number of stored states of about 23%.

The verification of property 3 took longer (21 minutes), required a greater search-depth to be reached (4.2 million) and more states to be explored (3.7×10^6). 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.

8 Features

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

8.1 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.
- **OCO – originating calls only** The subscriber is only able to be the originating party of a call.

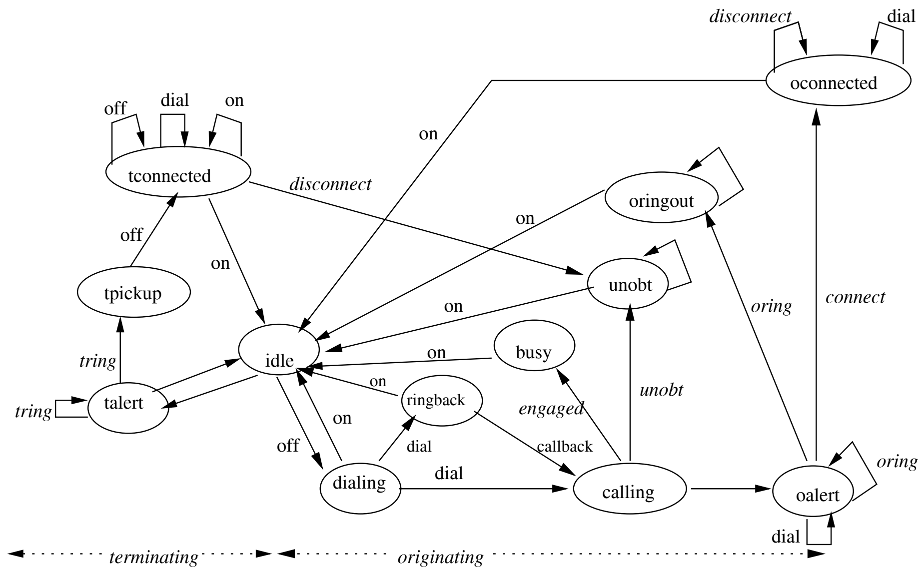

Fig. 5. Finite State Automaton for RBWF

- **TCO – terminating calls only** The subscriber is only able to be the terminating party of a call.

Note that examples of services that offer the features OCO or TCO are a pay phone and a teen line respectively.

We do not give automata for all the features, but give only one example. Figure 5 illustrates the change in user-perceived behaviour when the user subscribes to the ring back when free feature.

9 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 (see for example [14]). 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 13).

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 RBWF.

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 $event[i] == call$.

Property 13 – OCO Assume that user j has OCO.

No connection from user i to user j is possible.

That is: $[](\neg p)$, where p is $connect[i].to[j] == 1$.

Property 14 – TCO Assume that user j has TCO.

No connection from user j to user i is possible.

That is: $[](\neg p)$, where p is $connect[j].to[i] == 1$.

10 The Features in Promela

We do not give all the details of the implementation of features in Promela, but draw attention to some of the more important aspects:

- 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 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 reinitialisation).
- 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.

10.1 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. In SPIN, an inline definition is defined at the same level as a proctype declaration. An inline invocation (an inline call) is performed with the same syntax as a procedure call in an imperative language, such as C, and the parameters to an inline definition are typically names of variables. The body of an inline is expanded within the body of the User proctype at each point of invocation.

$feature_lookup$ is defined as follows:

```
inline feature_lookup(part_chan,part_id,st)
{
 do
 ::((st==st_idle)&&(term_call_only[selfid]==1))->st=st_blocked
 ::((st==st_diall)&&(ODS[selfid]==part_id))->st=st_unobt
 ::((st==st_diall)&&(RBWF[selfid]==1)&&(part_id==7))
         ->st=st_rback
 ::((part_id!=7)&&(st==st_diall)&&(CFU[part_id]!=6))
          ->part_id=CFU[part_id];
            part_chan=chan_name[part_id]
 ::((part_id!=7)&&(st==st_diall)&&(CFB[part_id]!=6)&&(len(part_chan)>0))
          ->part_id=CFB[part_id];
            part_chan=chan_name[part_id]
 ::((st==st_call)&&(OCO[part_id]==1))->st=st_unobt
 ::((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 13).

We note that one may regard $feature_lookup$ as encapsulating 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.

	BC	CFU	CFB	OCS	ODS	TCS	RBWF	OCO	TCO
property 1	√	×	√	×	×	×	√	×	×
property 2	√	×	×	√	√	√	√	×	√
property 3	√	√	√	√	√	×	√	×	√
property 4	√	×	×	√	√	√	√	√	√
property 5	√	×	×	×	√	×	√	×	√
property 6	√	√	√	√	√	√	×	√	√
property 7	×	√	–	–	–	–	–	–	–
property 8	×	–	√	–	–	–	–	–	–
property 9	×	–	–	√	–	–	–	–	–
property 10	×	–	–	–	√	–	–	–	–
property 11	×	–	–	–	–	√	–	–	–
property 12	×	–	–	–	–	–	√	–	–
property 13	×	–	–	–	–	–	–	√	–
property 14	×	–	–	–	–	–	–	–	√

Fig. 6. Results of Property validation

10.2 Feature Validation

Each feature was validated (via SPIN verification) against the appropriate set of properties (1–12). We consider the property associated with a feature, the basic call properties, and the other feature properties, as appropriate. In figure 6 a dash (–) denotes that the property is not appropriate – for example a property might depend upon data that is not relevant to that feature. A × indicates that for some suitable set of parameters, the property is not satisfied. For ease of presentation and for comparison, BC is considered (again) as a feature.

11 Static Analysis

Static analysis is an analysis of the *structure* of the feature descriptions, i.e. an examination of the Promela descriptions. In this analysis, interaction is defined as follows.

Definition An *interaction* is when *overlapping* guards: two or more guards which evaluate to true, under an assignment to variables, lead to diverging consequences.

In other words, if there is an overlap, and the consequences diverge, then there

is non-determinism and hence a potential interaction. A more operational explanation is that we are trying to detect of shared *triggers* of features. Shared trigger is a well known concept, though seldom expressed in the above way. Because we have collected additional feature behaviour together within the inline $feature_lookup$, we need only consider overlapping guards within this function.

As an 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 $x, y \neq 6$ and $x \neq y$. The first choice consequently assigns x to $part_id$, but 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.

Interactions found from static analysis are relatively rare, because the shared triggers indeed lead (for different reasons) to the same action, or because there are few overlaps. Overlaps (found through the process of superposition) are sometimes subtle. For example, consider the following choices:

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

There is no overlap here because 7 is not a valid number to be in a screening list, hence no interaction.

In all, there are 9 pairs to consider (4 clauses for st_dial, leading to 6 pairs, and 3 clauses for st_call, leading to 3 pairs). The results of the static analysis are given in the tables of figure 7 and 8. 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 depends very much on the structure of the specification, unlike our *dynamic* form of analysis.

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

Fig. 7. Feature Interaction Results - Static Analysis, SUSC

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

Fig. 8. Feature Interaction Results - Static Analysis, MUMC

12 Dynamic Analysis

Dynamic analysis is an analysis of the logical properties that are satisfied (or not) by pairs of users subscribing to combinations of features. In this analysis, an interaction is defined as follows.

Definition Let x and y be user processes, and $x_{f_i} \cup y_{f_j}$ the *configuration*, or *scenario*, in which x subscribes to feature f_i and y subscribes to feature f_j. 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, for a property ϕ, we have $x_{f_i} \models \phi$ but $x_{f_i} \cup y_{f_j} \not\models \phi$.

When $x == y$, then the interaction is SUSC, otherwise it is MUMC. Note that there are no constraints on i and j, ie. $i = j$ or $\neq j$. An SUSC (MUMC) interaction between f_i and f_j, resulting from a violation of property ϕ_i is

written $(f_i, f_j)_S$ $((f_i, f_j)_M)$.

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 [32]. A similar approach to dynamic analysis is taken, for example, by [39].

A naive approach would be to consider *any* property above as a candidate for ϕ. However, it is easy to see that this would lead to all features interacting. 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.

13 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
```

13.1 Dynamic Analysis – Feature Interaction results

The tables in figure 9 and 10 give the interactions found (using automated model generation and analysis) 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 (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 $(RBWF, CFU)_S$

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

Fig. 9. Feature Interaction Results - Dynamic Analysis, SUSC

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

Fig. 10. Feature Interaction Results - Dynamic Analysis, MUMC

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 13) 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 sequence 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.

14 Role of Static and Dynamic Analysis

The interactions found through 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 analysis.

We may regard the static analysis step as an inexpensive 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.

In order to illustrate the contributions of the two analyses better, consider two examples: when an interaction is detected and when it is not.

14.0.1 ODS and CFU

Static analysis detects an interaction when $ODS[0] = 1$, and $CFU[1] = 2$. Dynamic analysis also detects an interaction, with $i = 0$ and $j = 1$. In any trace where $feature_lookup$ takes the ODS branch, there is no interaction (the CFU property is not violated because we do not even get the stage of user 0 attempting to ring user 1). One could understand this as ODS having precedence. However, there is a trace where the CFU branch is taken; in this case CFU has precedence and the ODS property is violated because user 0 has *dialled* user 1 – before the call is forwarded to user 2.

14.0.2 OCS and CFU

Neither static nor dynamic analysis detects an interaction. In the static case, this is obvious because the OCS case refers to the *call* state and the CFU case refers to the *dial* state. These are disjoint states, hence no overlap. However,

one might expect to detect an interaction dynamically, thinking of the scenario described above ($OCS[0] = 1$, $CFU[1] = 2$, and user 0 rings user 1). It seems that one property must be violated. But, there is no violation and the clue as to why comes from the static analysis: CFU is triggered properly whilst in the *dial* state, but later on, in the *call* state, OCS stops the call progressing. The property for CFU is fulfilled, we do *attempt* to make the appropriate call, and then that call is blocked. So, both properties are satisfied and we do not have an interaction.

This example highlights again the fact that the properties are not complete descriptions, we do not insist that a connection is made for CFU. As before, it also illustrates why complete descriptions at this level are not appropriate.

It is also important to note that the particular specification detail of a feature can affect the interaction results associated with that feature. For example, had we chosen to implement the OCO feature from within the *dialing* state (rather than in the *calling* state), far more interactions would have exhibited. There would, for example, have been a static (MUMC) interaction between ODS and OCO and a dynamic (MUMC) interaction between CFA and OCO.

14.1 Comparison with Other Feature Interaction Results

Unfortunately, it is very difficult to compare interaction analysis results – even when researchers model the (apparently) same features, actual specification detail can profoundly affect the result (as seen above). Nevertheless, we have compared our results with those from [39] and the contest run in conjunction with FIW 2000 (*Feature Interaction Workshop 2000*) [3]. Our results are, broadly speaking, similar, though we note that our interpretation of $RBWF$ is rather different from that in the contest.

15 Conclusions and Future Directions

Our approach to feature interaction detection involves modelling a service at two different levels of abstraction: communicating finite state automata and temporal logic formulae, represented by Promela specifications, labelled transition systems and Büchi automata. In this paper, we have considered modelling and analysing a basic call service with eight features, involving four users with full functionality. There are two types of analysis, static and dynamic; the latter inolves model-checking with SPIN and is completely automated, making extensive use of Perl scripts to generate the SPIN runs.

The application area is a challenging one for model-checking for two reasons: formulating the right temporal properties for distributed systems is difficult, and the state spaces for any realistic model quickly become intractable. We have demonstrated why relativised properties are important in distributed systems and how they can be implemented in the LTL of SPIN, and how a Promela model can be optimised, without losing operational detail. For the latter, we have outlined a simple but effective state-space reduction 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 [39]. We believe that both our reduction technique and the use of Perl scripts could be useful to the SPIN community in general.

Since our analysis technique is based on property violation, and reasoning by model-checking always provides a counter-example, we can gain a good understanding of why an interaction occurs. Since our service model is low-level, we can quickly see which operational aspects (eg. local, global variables) are causing an interaction, this 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.

Finally, we observe that we have only proved our results for four user processes, we would like to be able to generalise these results for *any* number of user processes. This is currently under investigation.

Acknowledgements

The authors would like to thank Gerard Holzmann for his help and advice, and the Revelation project at Glasgow for providing computing resources. This work has been supported by a Daphne Jackson Fellowship Funded by the UK Engineering and Physical Sciences Research Council and by Microsoft Research.

References

[1] Dragon Bosnacki. Partial order reduction in presence of rendez-vous communication with unless and weak fairness. In *[11]*, pages 40–57, 1999.

[2] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems.* IOS Press (Amsterdam), May 1994.

[3] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems*, volume VI. IOS Press, Amsterdam, 2000.

[4] 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.

[5] M Calder and A Miller. Using SPIN for feature interaction analysis - a case study. In *[15]*, pages 143–162, 2001.

[6] Muffy Calder and Alice Miller. Analysing a basic call protocol using Promela/XSpin. In *[25]*, pages 169–181, 1998.

[7] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for IN and beyond. In *[2]*, pages 1–23, May 1994.

[8] Yaacov Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

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

[10] Costas Courcoubetis, editor. *Proceedings of the Fifth International Conference on Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, Elounda,Greece, June/July 1993. Springer-Verlag.

[11] 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.

[12] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[13] P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV.* IOS Press (Amsterdam), June 1997.

[14] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second International Workshop on Formal Methods in Software Practice (FMSP '98)*, pages 7–15. ACM Press, March 1998.

[15] M.B. Dwyer, editor. *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, Toronto, Canada, May 2001. Springer-Verlag.

[16] K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *[22]*, pages 236–248, 1999.

[17] A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In *[3]*, pages 179–192, May 2000.

[18] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th international Conference on Protocol Specification Testing and Verification (PSTV '95)*, pages 3–18. Chapman & Hall, Warsaw, Poland, 1995.

[19] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[20] Susanne Graf and Claire Loiseaux. A tool for symbolic program verificaion and abstration. In *[10]*, pages 71–84, 1993.

[21] The Grid Forum. `http://www.sdsc.edu/GridForum/RemoteData/papers/gridft_intr_gf5.pdf`.

[22] 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.

[23] Constance L. Heitmeyer, James Jr. Kirby, Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11), November 1998.

[24] 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.

[25] 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.

[26] Gerard J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25:981–1017, 1993.

[27] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[28] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *[24]*, pages 197–211, 1994.

[29] 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.

[30] *IN Distributed Functional Plane Architecture*, recommmendation q.1204, ITU-T edition, March 1992.

[31] K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.

[32] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In *[3]*, pages 311–325, May 2000.

[33] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical Report STAN-CS-90-1321, Stanford University, June 1990.

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

[35] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996.

[36] Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *[38]*, pages 233–257, 1996.

[37] Doron Peled, Thomas Wilke, and Pierre Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and ω-regular languages. *Theoretical Computer Science*, 195:183–203, 1998.

[38] Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors. *Proceedings of the DIMACS Workshop on Partial-Order Methods in Verification (POMIV '96)*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[39] M. Plath and M. Ryan. Plug-and-play features. In *[31]*, pages 150–164, 1998.

[40] A.W. Roscoe. Model-checking csp. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 21, pages 353–378. Prentice-Hall International, 1994.

[41] M. Thomas. Modelling and analysing user views of telecommunications services. In *[13]*, pages 168–182, 1997.

Appendix: The Basic Service with features

We give here the optimised code for the basic service plus features. All code pertaining to feature behaviour is commented out.

```
mtype =
{on,off,dial,call,oring,tring,unobt,engaged,connected,disconnect,
callback,st_idle,st_blocked,st_unobt,st_rback,st_diall,st_call};

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 number to channel name */

/*byte call_forward_always[4];*/ /*the ith member of these arrays switched to */
/*byte call_forward_busy[4]; */  /*default value of 6 if user[i]*/
/*byte orig_call_screen[4];*/  /*does not have this feature,*/
/*byte orig_dial_screen[4];*/   /* and to identity  of user that */
/*byte term_call_screen[4];*/   /*user[i] forwards to, or cant call */
                                /* or cant be called by, otherwise*/

/*byte ring_back_when_free[4];*/ /*set to 0 or 1*/
/*byte orig_call_only[4];*/ /*set to 0 or 1*/
/*byte term_call_only[4];*/ /*set to 0 or 1*/

/*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=-1;*/
   /*p3= -1;*/
   /*p1= -1,p2=-1*/

   /* The simple basic call protocol: A rings B */
   /* A goes offhook, put A,0 on channel A   */
   /* A dials B and B is free,  */
   /* then A puts A,0 on channel B; B,0 on channel A.           */
   /* B goes offhook and then put B,1 on channel A         */
   /* A and B are now connected */
   /* To disconnect: A removes token from own channel,  */
   /*  B removes token from own channel  */

/*inline feature_lookup (q1,id1,st)

{
 do
 ::((st==st_idle)&&(term_call_only[selfid]==1))->st=st_blocked
 ::((st==st_diall)&&(orig_dial_screen[selfid]==id1))->st=st_unobt
 ::((st==st_diall)&&(ring_back_when_free[selfid]==1)&&(id1==7))->st=st_rback
 ::((id1!=7)&&(st==st_diall)&&(call_forward_always[id1]!=6))
    ->id1=call_forward_always[id1];
      q1=chan_name[id1]
 ::((id1!=7)&&(st==st_diall)&&(call_forward_busy[id1]!=6)&&(len(q1)>0))
    ->id1=call_forward_busy[id1];q1=chan_name[id1]
```

```
 ::((st==st_call)&&(orig_call_screen[selfid]==id1))->st=st_unobt
 ::((st==st_call)&&(term_call_screen[id1]==selfid))->st=st_unobt
 ::((st==st_call)&&(orig_call_only[id1]==1))->st=st_unobt
 ::else->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=null;
 bit messbit=0;
 mtype state=on;
mtype dev=on;
 byte partnerid=6;

idle:
atomic{
    assert(dev == on);
    assert(partner[selfid]==null);
     /* either attempt a call, or receive one */
        if
    :: empty(self)->state=st_idle;feature_lookup(partner[selfid],partnerid,state);
     if
    :: state==st_blocked->state=on;goto idle
    :: else->state=on
    fi;
    /*event_action(off);*/

       dev=off;
       self!self,0;goto dialing
          /* 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 preidle
               fi
    ::empty(partner[selfid])->
      self?messchan,messbit;  /* call attempt cancelled */
      partner[selfid]=null;partnerid=6;messchan=null;messbit=0;
      goto preidle
    fi
```

```
        fi};

dialing:
atomic{
      assert(dev == off);
      assert(full(self));
      assert(partner[selfid]==null);
       /* dialing or go onhook */
        if
:: /*event_action(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;
      state=st_diall;
     /* feature_lookup(partner[selfid],partnerid,state);*/
 if
      ::state==st_unobt-> state=on;partner[selfid]=null;partnerid=6;/*dialled[selfid]=6;*/
        goto unobtainable
      /*::state==st_rback-> state=on;goto ringback*/
      ::(state==st_diall&&partnerid!=7)-> state=on;goto calling
      ::(state==st_diall&&partnerid==7)-> state=on;partner[selfid]=null;partnerid=6;
        /*dialled[selfid]=6;*/
        goto unobtainable
 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);
    assert(full(self));
   /* record[partnerid]=selfid;*/
    state=st_call;
    /*feature_lookup(partner[selfid],partnerid,state);*/
    if
    ::state==st_unobt->state=on;partner[selfid]=null;partnerid=6;/*dialled[selfid]=6;*/
    goto unobtainable
    ::state==st_call->state=on;skip
    fi;
    if
    :: partner[selfid] == self -> goto busy
            /* invalid partner */
            :: partner[selfid]!=self ->
               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 or trivial dial */
   atomic{
  assert(full(self));
  /*network_ev_action(engaged);*/
      if
  :: /*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 */
 /* :: event_action(dial);
      goto busy*/
         /* trivial dial */
      fi};

 /*comment out entire ringback state when no ACB switched on */
/*otherwise just comment out events when not needed */
/*Can't nest comments remember*/
/*ringback:
  atomic{printf("MSC: Last Caller was user %d\n",record[selfid]);
          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 is unobtainable, go onhook or trivial dial */
      atomic{assert(full(self));
             assert(partner[selfid]==null);
             assert(partnerid==6);
            /* assert(/*dialled[selfid]==6);*/
           /*network_ev_action(unobt);*/
      if
         ::/*event_action(on);*/
         dev = on;
                self?messchan,messbit; assert (messchan==self);
                messchan=null;
        messbit=0;goto preidle
         /*go onhook, cancel connection attempt */
         /*::event_action(dial);goto busy*/
         /* trivial dial */
      fi};


oalert:
/* called party is ringing */
atomic{
```

```
       assert(full(partner[selfid]));
       assert(full(self));
assert(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 */
/* :: event_action(dial);messbit=0;goto oalert*/
           /* trivial dial */
        fi};


oringout: /*abandon call attempt*/
         atomic{
assert(full(partner[selfid]));
 assert(full(self));
 assert(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->skip /* attempt has been cancelled */
            fi;
        /*network_ev_action(disconnect);*/
        self?messchan,messbit;
        partner[selfid]=null;partnerid=6;
        /*dialled[selfid]=6;*/
        messchan=null;messbit=0;
        goto preidle
  };


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) -> /* trivial handset down */
       /*event_action(on);*/
       dev = on; messchan=null;messbit=0;goto tclose

               ::(messbit == 1 && dev == on) ->  /* trivial handset up */
```

```
       /*event_action(off);*/
       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 */


init
{
 atomic{partner[0]=null;
partner[1]=null;
partner[2]=null;
partner[3]=null;
chan_name[0]=zero;
chan_name[1]=one;
chan_name[2]=two;
chan_name[3]=three;

 /*switch on features here*/
 /*default value 6, */
 /*if user i has feature, set to id of user to be forwarded to, or screened */
/*call_forward_busy[0]=6;
call_forward_busy[1]=6;
call_forward_busy[2]=6;
call_forward_busy[3]=6;*/

/*call_forward_always[0]=1;
call_forward_always[1]=6;
call_forward_always[2]=6;
call_forward_always[3]=6;*/

/*orig_dial_screen[0]=6;
orig_dial_screen[1]=6;
        orig_dial_screen[2]=6;
orig_dial_screen[3]=6;*/

/*orig_call_screen[0]=6;
orig_call_screen[1]=6;
orig_call_screen[2]=6;
orig_call_screen[3]=6;*/

/*term_call_screen[0]=6;
```

```
term_call_screen[1]=6;
term_call_screen[2]=6;
term_call_screen[3]=6;*/

/*ring_back_when_free[0]=0;
ring_back_when_free[1]=0;
ring_back_when_free[2]=0;
ring_back_when_free[3]=0;*/

/*orig_call_only[0]=0;
orig_call_only[1]=0;
orig_call_only[2]=0;
orig_call_only[3]=0;*/

/*term_call_only[0]=0;
term_call_only[1]=0;
term_call_only[2]=0;
term_call_only[3]=0;*/


      /*p0=*/run  User(0,zero);
      /*p1=*/ run  User(1,one);
/*p2=*/ run  Orig_User(2,two);
/*p3=*/ run  Term_User(3,three);
     }
}
```