Feature Interaction Detection by Pairwise Analysis of LTL Properties - a case study

Muffy Calder and Alice Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.
muffy,alice@dcs.gla.ac.uk

Abstract A Promela specification and a set of temporal properties are developed for a basic call service with a number of features. The properties are expressed in the logic LTL.

Interactions between features are detected by pairwise analysis of features and properties. The analysis quickly results in both state-space and property case explosion. To overcome this state-spaces are minimised, model checking results generalised through symmetry and bisimulation, and analysis performed automatically using scripts. The result is a more extensive feature interaction analysis than others in the field.

Keywords communicating processes, distributed systems, model checking, feature interaction, communications 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, by different developers. A consequence of adding features in this way is *feature interaction*, where the behaviour of one feature modifies the behaviour of another, often leading to unpredicted and/or undesirable results. The problem is a long standing one within the communications services domain [37, 3] and exhibits in many other component-based domains. Detection of interactions is crucial to any solution (whether system redesign or a run-time solution), but in complex systems, especially when there is a proliferation of features, detection by manual inspection is not feasible. Automated techniques are therefore essential. In this paper we use a combination of model checking, symmetry and scripting for detecting interactions.

After introducing the preliminaries in Section 2, the paper has two parts.

First, in Sections 3 to 6, we develop a specification in Promela [26, 30] for a basic call service. Promela is an appropriate language because the service is inherently concurrent, with *asynchronous* communication. We develop a set of temporal properties in order to validate the specification. We discuss how to express the properties in the linear temporal logic LTL, and how to verify them

using the model checker SPIN[1] [27], paying particular attention to state-space reduction.

Second, in Sections 7 to 10, we develop a feature set and perform pairwise interaction analysis. The analysis is completely automated, making extensive use of Perl scripts to generate the model checking runs. In section 11 we introduce a method based on abstraction and induction whereby we show how some of our results can be generalised for systems consisting of any number of users, in which at most two users have features.

We compare our results with others in the field in Section 12 and present our conclusions in Section 13. Some preliminary results have been presented earlier [4]; our analysis here is more extensive because we consider 9 features whereas previously 6, less complex, features were considered. We also employ symmetry, generalisation and include greater implementation detail.

2 Background

2.1 Communications Services and Feature Interactions

Our concern is the control of calls between two parties, the actual data exchanged (e.g. audio or digital) is not of interest. Control is provided by a *service*, in classical telecommunications, this is provided by 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 *feature* is additional functionality, for example, a *call forwarding capability*, or *ring back when free*; a user is said to *subscribe* to a feature. An interaction is a behavioural modification between two or more features and/or a service.

As an example, consider a user who subscribes to *call waiting* (CW) and *call forward when busy* (CFB) and is engaged in a call. What will happen when there is a further incoming call? (Full details of all features mentioned here are given in section 7.) If the call is forwarded, then the CW feature is clearly compromised. If, on the other hand, call waiting is activated, then the CFB feature is compromised. Clearly both features cannot proceed as they would in isolation. This interaction is relatively simple (e.g. it can be seen as inconsistent requirements) because both features are subscribed to by a single user. We refer to this situation as a single user (SU) interaction. More subtle interactions can occur when more than one user/subscriber are involved, these are referred to as multiple user, (MU) 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 required by B's feature CFB), then A's feature OCS is compromised. On the other hand, if the call is not forwarded, then B's CFB feature is compromised. These kind of interactions (i.e. MU) can be very difficult to detect (and resolve), particularly since different features may be activated at different stages of a the call cycle.

[1] Unless stated otherwise, we use SPIN version 4.07 throughout this paper.

Interactions may be characterised informally as type I or type II [24]. Interactions which arise from inconsistent specifications, e.g. inconsistent state changes or inconsistent events, are called type I. These interactions are usually the result of a "shared trigger", for example, in the case of CW and CFB above, both features are triggered by an incoming call. Type II interactions do not involve a shared trigger but still result in inconsistent user intentions, as demonstrated by the interaction above between OCS and CFB. Type II interactions can only be detected with reference to user intentions, i.e. properties of features; they are the primary concern of this paper.

2.2 Call Control and Feature Reference Models

There are numerous call models and feature sets in the literature. Since our motivation is detecting type II interactions, we take a user perspective, following the *IN* (*Intelligent Networks*) model, distributed functional plane [34]. Our basic call model is adapted from this standard, as follows.

Basic Call We implement the *IN* BCSM (basic call state model) with minor amendments:

– We unify the two null states (originating and terminating) *O_Null* and *T_Null* into the state *O/T_Null*.
– Since routing is not our concern (we assume a single network), we merge the states *Auth_Orig_Att*, *Collect_Info*, *Analyse_Info*, *Select_Route* and *Auth_Call_Setup* into the single state *Auth_Orig_Att*.
– We separate the *O_Exception* state into two states, depending on the trigger events *noanswer* and *busy*, thus the states are *O_No_Answer* and *O_Busy*.
– We make call tear down asymmetric. This reflects the behaviour of the UK PSTN and results in behaviour which is potentially more interesting than a symmetric tear down.

The BCSM is only a state model, there are no intentions, or explicit properties given in this standard. We therefore introduce a set of properties based on practical experience and properties presented in the feature interaction literature [37, 3].

Features It is more difficult to select a standard feature set. Although the *IN* Capability Set 1 (CS-1) [34] enumerates a number of features, behaviour is not defined. Most detailed descriptions of commercial features are proprietary. We have therefore chosen a feature set which is based on published literature, namely the feature set used in an SMV-based study [48]. Details of the feature set are given in section 7; comparison of this work with the SMV study is given in section 12.

2.3 Promela and Spin

Promela, *Pro*cess *me*ta *la*nguage [26, 27, 30], 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 rendezvous (synchronizing) communication, dynamic process creation, and mobile connections, i.e. communication channels can be passed along other communication channels. The language is very expressive, and has intuitive Kripke structure semantics.

Definition 1. *Let AP be a set of atomic propositions. A Kripke structure over AP is a tuple $\mathcal{M} = (S, S_0, R, L)$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation and $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

From here on we assume that $|S_0| = 1$, i.e. there is a single initial state, s_0 say.

Spin is a bespoke model checker for Promela and provides several reasoning mechanisms: assertion checking, acceptance/progress states and cycle detection and satisfaction of temporal properties. These properties are expressed in temporal logic [41, 17, 16]. The logic used is *linear temporal logic* LTL [49]. When performing verification, we use Xspin, the graphical interface for Spin.

We note that in the context of model checking the terms "model", "specification" and "system" are easily confused. Here we try to adopt the following convention: a model is Kripke structure, a specification is a Promela description (from which a Kripke structure is derived), and a system is a commonly understood physical implementation or abstraction thereof. We note that there are further uses of these terms, e.g. the basic call model.

Other popular model checkers include SMV [45], Murphi [13], FDR [50] and the Java PathFinder model checker JPF [55]. We choose to work with Promela and Spin primarily because of the rich, expressive power of Promela, in particular, asynchronous communication. Spin has been widely used to model check protocols associated with software-controlled systems, for example, in communications systems [5, 4, 32] and in railway interlocking systems [8].

2.4 Reasoning in Spin

In order to perform verification on a specification, Spin translates each process template into a finite automaton and then computes an asynchronous interleaving product of these automata to obtain the global behaviour of the concurrent system. This interleaving product is referred to as the *state-space*.

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 [27, 22, 44]. Checking satisfaction of a formula involves the depth-first search of the synchronous product of the automaton corresponding to the concurrent system (model) and the Büchi automaton corresponding to the never-claim.

Note that Büchi automata are more expressive than LTL itself (there are behaviours that can be expressed directly as Büchi automata that can not be expressed as LTL formulae). However, we find LTL to be quite adequate to describe the behaviour of our system, and prefer to allow SPIN's excellent LTL converter to create the Büchi automata for us. Other formalisms exist to express behaviour (e.g. timelines [53], or Message sequence charts [43]). We do not use either of these methods here. We do use the MSC illustration of counterexamples provided by SPIN however.

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. 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 XSPIN'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 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 properties (section 5.2), we use the shorthand $E\langle\rangle p$ (meaning *for some path* $\langle\rangle p$) to mean "($\langle\rangle p$ for No Executions) is not true".

2.5 Parameters and Further Options used in SPIN Verification

When performing verification with SPIN three numeric parameters must be set. These are *Physical Memory Available*, *Estimated State-Space Size* and *Maximum Search Depth*. The meaning of the first of these is clear, and the second controls the size of the state-storage hash table. The *Maximum Search Depth* parameter determines the size of the *search-stack*, where the states in the current search are stored. If comparisons are to be made with other model checkers, then the value of the Maximum Search Depth should be taken into account because its value determines the size of the stack provided for state storage, and so affects the total memory used. For this reason, in our verification results (see section 6.3 for example) we give only the memory required for state storage, and not the total memory required.

Partial order reduction (POR) [47, 46] is based on the observation that execution sequences (or "traces") can be divided into equivalence classes whose members are indistinguishable with respect to a property that is to be checked. We apply POR in most cases.

Compression (COM) [28] is a method by which each individual state is encoded in a more efficient way. We apply compression in all cases.

Weak Fairness (WF) is a constraint which ensures that the only paths considered are those in which any process that has has an enabled transition will eventually do so. The use of WF is expensive, as it involves several copies of the state space being maintained, and so we avoid its use whenever possible (see section 9).

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

3 Basic Call Service

In the next section we give an overview of the Promela specification. The specification is quite detailed and so by way of introduction, in this section we give a more abstract behaviour description.

Figure 1 gives a diagrammatic representation of the automaton for the basic call service. States to the left of the *O/T_Null* 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 *O/T_Disconnect* 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 *Call_Sent* to *O_Alerting*, simply because there is no *observable* event associated with this transition.

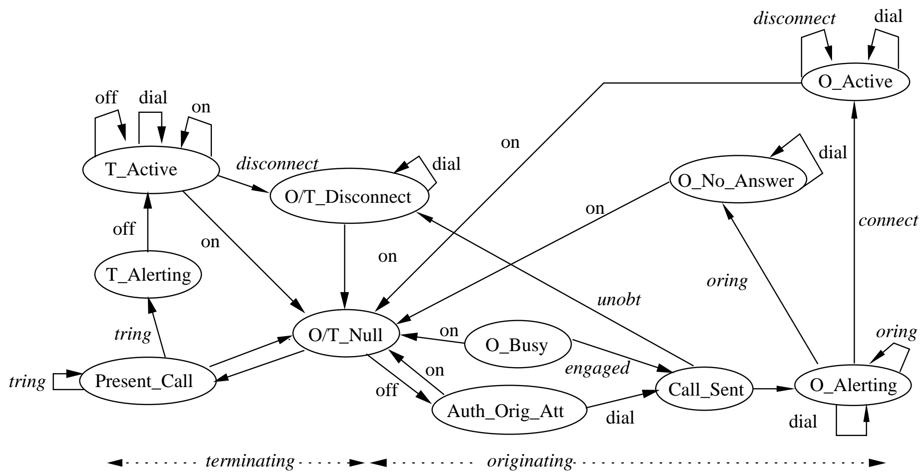

Figure1. Basic call - states and events

The automata must communicate with each other; the behaviour of one call process, as originating party, influences the behaviour of another call process, as terminating party. Since our motivation here is explanation rather than rigorous specification (that is the role of the Promela specification), we do not extend the automata to include a communication mechanism, rather we describe the communication informally.

A communication channel is associated with each call process. Each channel has capacity for at most one message: a pair consisting of a channel name (that associated with itself or the other party in the call) and a status bit (the status of the connection). When it is not confusing, we refer to the 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 2.

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 dialing 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 is changed to 0. Then, since both A and B have status bit 0, neither process is 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.

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

Figure2. States of a communication channel in the protocol

4 Basic Call Service in Promela

Each call process (see figure 1) 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, therefore, we represent events by (their effect on) event variables (e.g. $event[i]$ or $network_event[i]$) and call states (e.g. *Call_Sent, Auth_Orig_Att*, etc.) by labels. Since each transition is implemented by several (possibly compound) statements, we group these together as an *atomic* statement, concluding with the appropriate *goto*.

An example of the Promela code associated with the *O/T_Null*, *Auth_Orig_Att*, *Call_Sent* and *O_Active* states and their outgoing transitions is given below. The global/local variables and parameters include the self-explanatory `selfid` and `partnerid`, the communication channel associated with the specific process `self`, the `partner` array, recording the channel name of the current partner of each process, the arrays *connect.to*, recording the presence of a connection between two users, the local variable `dev` recording the current status of the device (*on* or *off*), the `dialed` array recording the most recent number dialed (since leaving the *O/T_Null* state) for each process, and the `event` and `network_event` events recording the most recent user-initiated and network-initiated events of each process, respectively. In addition `messchan` and `messbit` are local variables used for reading messages. The channel `null` allows a default value for any of the `partner` variables when the corresponding call process is not engaged in a call. This value is not strictly necessary for modelling purposes, but can be valuable for reasoning. Note that we use 6 as a default value (for the `partnerid` variable for example) of variables which can take the value of any process id.

Any variable about which we may intend to reason should not be updated more than once within any atomic statement (so that each change to the variables is visible to the never-claim), other variables may of course be updated as required. For this reason we have introduced a new state, *O_Close* so that we can monitor when a connection has been made successfully. Thus the connection is established within the *O_Active* state, via the setting of the array element $connect[selfid].to[partnerid]$ to 1, and the remaining behaviour usually associated with the *O_Active* state is contained within *O_Close*. (The full code for the *O_Close* state may be found in Appendix 1.) 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.

```
O/T_Null:
  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 Auth_Orig_Att
     /* 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 Present_Call
              :: else -> self?messchan,messbit;
     /* call attempt cancelled */
                 partner[selfid]=null;partnerid=6;
                 messchan=null;messbit=0;goto O/T_Null
              fi
            ::empty(partner[selfid])->
              self?messchan,messbit;
     /* call attempt cancelled */
              partner[selfid]=null;partnerid=6;
              messchan=null; messbit=0;
              goto O/T_Null
            fi
        fi};


Auth_Orig_Att:
  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; dialed[selfid] = 0;partnerid=0
                    :: partner[selfid] = one; dialed[selfid] = 1; partnerid=1
                    :: partner[selfid] = two; dialed[selfid] = 2; partnerid=2
                    :: partner[selfid] = three; dialed[selfid]= 3; partnerid=3
                    :: partnerid= 7;
                fi

                 :: event[selfid]=on; dev[selfid]=on;
                    self?messchan,messbit;assert(messchan==self);
                    messchan=null;messbit=0;
            goto O/T_Null
                   /*go onhook, without dialing */
         fi};


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

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

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)}
```

5 Basic Call Service Properties

In this section we give our set of temporal properties for the basic call service, in English, and their implementation as LTL formulae. Before doing so, we explain the form of the propositions.

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 $user[proci]@O/T_Null$, meaning the incarnation of the process *user* with process identifier *proci* is at label O/T_Null. Process identifiers are simply global variables, initialised when a process is instantiated (and captured by assignment within the Promela *run* command).

The variables referred to in our propositions include those described in section 4. Note that in addition *proci* and $chan_name[i]$ are the process identifier and the channel name associated with user process i, respectively.

5.2 Basic Call Service Temporal Properties

In [4] we showed how a *relativised next* operator (that is the next state *relative* to a particular constituent process) can be implemented in Spin. This was done by judicious use of the built-in global variable *_last* (a variable holding the value of the (internal) process number of the process that last made a transition) and the (LTL) *next* operator $\circ$. The availability of such an operator is helpful in allowing us to express a greater number of properties. However, since the use of the *_last* variable within a property precludes the use of partial order reduction, the usefulness of the *relativised next* operator is restricted. Therefore we no longer use this operator and rely instead on the operators $\mathcal{W}$ (*weak* until) and $\mathcal{P}$ (precedes), defined as follows:

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

and

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

As described in section 2.4 we use the shorthand notation $E\langle\rangle p$ (*for some path p*) to mean "($\langle\rangle p$ for No Executions) is not true".

The LTL is given here alongside each property. This involves referring to variables (e.g. *dialed* and *connect.to*) contained within the Promela code (an extract of which is given in section 4). We use symbols to denote propositions, and give our properties in terms of these symbols. An example might be "$[]p$ where p is $dialed[i] == i$". This provides a neater representation, and the LTL converter requires properties to be given in this way.

It is often necessary to refer to the *particular point in the service* reached by a process. For the basic call properties it is particularly important to monitor when an attempted call is completed, for example, and a process returns to the *O/T_Null* state. We do this via a statement of the form $user[proci]@O/T_Null$. In a similar way we can identify the position that $process[i]$ has reached in the service by the value of one of its corresponding event variables (that is $event[i]$ or $network_event[i]$). In some cases it is necessary to use the program position (via a suitable @ statement) and in others it is more suitable to refer to the value of an event variable. The particular property dictates which to use. For the basic call properties it is very straightforward which type of statement to use. However, in section 9 we show how, in more complicated properties, the choice can be less obvious.

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

LTL: $E\langle\rangle p$

$p = (connect[i].to[j] == 1)$, for $i \neq j$.

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

LTL: $[](p \rightarrow ((\neg r)\mathcal{W}q))$

$p = (dialed[i] == i)$, $q = (network_event[i] == engaged)$,
$r = (user[proci]@O/T_Null)$.

Property 3 *Busy tone or ringing tone will follow calling.*

$[](p \rightarrow ((p\mathcal{W}q) \vee (p\mathcal{W}r)))$

$p = (event[i] == call)$, $q = (network_event[i] == engaged)$,
r= $(network_event[i] == oring)$.

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

LTL: $[](p \rightarrow q)$

$p = (dialed[i] == j)$, $q = (partner[i] == chan_name[j])$.

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 O/T_Null state.*

LTL: $[]((p \wedge v \wedge t) \rightarrow (((\neg s)\mathcal{W}(w)) \vee ((\neg r)\mathcal{W}q)))$

$p = (dialed[i] == j)$, $v = (event[i] == dial)$, $t =(full(chan_name[j]))$,
$s = event[i] == call$, $w =(len(chan_name[i]) == 0)$,
$r = user[proci]@O/T_Null$, $q = (network_event[i] == engaged)$, 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). The reason that SPIN prevents the negation of *empty* and *nfull* is that they are statically determined as *safe* operations with respect to partial order reduction [31]. As *len* is not marked statically as safe, no such restriction arises.

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

LTL: $[](p \rightarrow q)$

$p = (user[proci]@Call_Sent)$, $q = (event[i] == dial)$.

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.

5.3 Property Refinement and Specification Patterns

There are two common problems which may arise due to the improper use of LTL, namely that invalid results may be obtained or that an unwarranted increase in the complexity of a verification run may result [29]. Great care has therefore been taken to ensure that each temporal formula not only expresses a property precisely, but that the formula will enable us to reason about our model in the most efficient way. It may therefore be necessary to take a series of *refinement steps* to ensure that our property is expressed correctly. For example, it would be tempting to express Property 2 as $[](p \rightarrow \langle\rangle q)$, where p is

($dialed[i] == i$) and q is ($network_event[i] == engaged$) (see [5]). This formula would be problematic in two ways. On the one hand it could be satisfied in a situation where a caller dialed his/her own number but failed to hear the engaged tone as a result (but heard the engaged tone *ultimately*, during a different call). This would result in no error being reported when most likely the intention was that the *scope* of the $\langle\rangle$ operator should extend *only* to the point at which the handset is replaced. On the other hand, this formula would cause an error to be reported if a caller dialed his/her number and then simply *failed to progress* infinitely often. To avoid this unwanted scenario, the weak-fairness option would be required, which involves a multiplication in the size of the state-space by a factor of N, where N is the number of processes, so causing a huge increase in the search depth/time. The use of the $\mathcal{W}$ operator in this situation is therefore crucial to limit the *scope* of the property. Note that the *dialed* and *network_event* elements associated with $User[i]$ are reset to their default values when $User[i]$ returns to the idle state. Therefore they only record events that have occurred within the current call.

Refinement involves checking suspicious results (by performing simulation runs, and closely examining error trails for example) and modifying the properties if necessary. It is also vitally important to examine the never claim (Büchi automaton) generated for the LTL formula. Sometimes examination of the never claim alone can illustrate that the LTL formula does not express the desired behaviour. (See section 6.2 for a further discussion of never claims.)

Some of our properties (especially the feature properties, see section 9) are highly complex and have taken many refinement steps to produce. The specification patterns of Dwyer et al [14] provide a useful way of creating LTL formulae from short template descriptions. As our properties have been developed over a number of years, we did not use the pattern specification system to a large degree for their formulation although it was this work that first alerted us to the fact that problems of *scope* are common. Some of the patterns that appear in our properties clearly adhere to the patterns described in [14] which provides both reassurance, and an easier way to construct properties in the future.

6 Basic Call Service Validation

For all verification runs described in this and subsequent sections, we used a PC with a 2.4GHz Intel Xenon processor, 3Gb of available main memory, running Linux (2.4.18).

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 Promela code itself can be transformed to optimise the state-space. The fully optimised code (including features) is given in Appendix 1.

6.1 Spin Options

The most obvious, standard optimisation to apply is POR. When applied, it does reduce the size of the state-space of our model (see section 6.3), but could our model be adapted to take further advantage of POR? Closer examination shows this to not be the case. 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 *xr*. 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). We choose to use only COM. Although MA and COM combined give a significant memory reduction, the trade-off in terms of time was simply unacceptable. For example, during initial attempts to verify property 2 using COM and MA, after 65 hours the depth reached was only of the order of 10^6.

6.2 State-space Management

There are several well-known strategies for reducing the size of the state-space of a model, one of these is the resetting of local variables. In our model this involves ensuring that each visit to a *call* state is indeed a visit to the same underlying Promela state. 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 *O/T_Null*. 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 *O/T_Null* 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 section 4). 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 *O/T_Null*. Therefore every occurrence of *goto* O/T_Null (within a state in which either of these variables are modified) 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. To overcome this problem, we use an *inline* function. In SPIN an inline function is a parameterised procedure with dynamic bindings. The body of an inline is expanded within the body of the User proctype at each point of invocation. Here the *event_action*(*eventq*) inline 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 *event_action* 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.) Notice that this reduction is only possible because the *event* (and similarly the *network_event*) variables are independent of each other – a change to the value of $event[i]$ say, does not effect (either directly or indirectly) the value of $event[j]$ where $i \neq j$.

We also note that an automatic (but conservative) application of this reduction is achieved via *slicing* (see for example [15]), which has been included as an integral part of the more recent versions of SPIN (since version 3.4.1). The slicing algorithm alerts the (SPIN) user to portions of the model (processes, statements, and data objects) that can be omitted without risk, and with a potential reduction in verification complexity.

These transformations not only lead to a *dramatic* reduction of the underlying state-space, the search depth required is 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 [10], [23] and [25]) these techniques are simple, and merely involve making simple checks that unnecessary states have not been unintentionally introduced. All SPIN users should be aware that they may be introducing spurious states when coding their problem in Promela. In [52] convenient "recipes" are provided to optimise both modelling and verification when using SPIN. Although some of the techniques described in [52] equate to our methods described above, we have not fully exploited other

useful suggestions contained therein. For example, see section 8 for a discussion on the use of bitvectors.

6.3 Verification Results

Using XSPIN it was possible to verify all six properties for four users 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. The validity of each property was reported via a *never-claim violation* message in the case of property 1 and via a *errors:*0 message for all other properties.

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 $2, 3, 4, 5$ and 6, when partial-order reduction was applied each search was completed within a maximum search depth of 3 million and there are at most 1.3 million stored states in each case. 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%. In table 1 below we give details of the verification of properties $2, 3, 4, 5$ and 6 (with POR) for the case $i = 0$ and $j = 1$ (if appropriate) where:

Depth describes the length of the longest path explored during the search
States is the number of states stored
Mem is the memory used (in Mbytes) for state-storage (with compression)
Time is the time taken (in seconds) = user time + system time and
state-vector is the size, in bytes, of the state-vector.

Notice that the size of the state-vector gives an indication of the number of variables that are required to be included within the model for the proof of the property.

Table1. Verification Results – basic call properties

Property	Depth ($\times 10^6$)	States ($\times 10^5$)	Mem	Time	state-vector
2	2.8	16	64.5	85	128
3	3.0	17	67.6	94	124
4	1.3	7.9	31.7	37	120
5	3.6	17	66.4	93	132
6	1.7	9.7	38.6	49	124

7 Features

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

7.1 Features

The set of features that we have added to the basic call include:

- **CFU – call forward unconditional** All calls to the subscriber are diverted to another user.
- **CFB – call forward when busy** All calls to the subscriber are diverted to another user, if and when the subscriber is busy.
- **OCS – originating call screening** All calls by the subscriber to numbers on a predefined list are prohibited. Assume that the list for user x does not contain x.
- **ODS – originating dial screening**. The dialing of numbers on a predefined list by the subscriber is prohibited. 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 prohibited. Assume that the list for user x does not contain x.
- **RBWF – ring back when free** If the subscriber is the originating party in a call to a busy line, a connection (from the subscriber to the other party) is reattempted when the terminating party becomes available. Assume that the subscriber is not both the originating and terminating party.
- **RWF – return when free** If the subscriber is the terminating party in a call to a busy line, a connection from the subscriber to the other party is attempted when the terminating party becomes available. Assume that the subscriber is not both the originating and terminating party.
- **OCO – originating calls only** The subscriber is only able to be the originating party of a call.
- **TCO – terminating calls only** The subscriber is only able to be the terminating party of a call.

As discussed earlier, we base our feature on the set given in [48]. The features CFU, CFB, RBWF, TCS, and OCS, and the associated properties, are well known and appear in [48]. We omit three features from [48]: CW (call waiting), because we restrict to one "leg" calls, CFNR (call forward no reply), because with respect to interaction analysis there is little difference between CFNR and CFB, and ACB (automatic call back) because it results in no interactions [48]. We add four further features, and associated properties: ODS, OCO, TCO, and RWF. ODS is based on informal discussions with telecomms providers – this is the feature that many people want when they invoke OCS, they don't care if a connection is made to a number, they just don't want to be *billed* for it. Hence there is a block on dialling the number, not on the connection. OCO and TCO

are features for a pay phone and (a form of) teen line. TCO is popular in the UK where calls are billed per connection *time*. RWF is a form of RBWF situated at the terminating side, it has also been called AR (automatic recall) [7].

We do not give automata for all the features, but give only one example. Figure 3 illustrates the change in user-perceived behaviour when the user subscribes to the ring back when free feature (RBWF). Note that the *set_ringback* and *unset_ringback* events correspond to the storing of the number of the subscriber's current partner (within an array) so that a ringback to that number can subsequently be initiated. The automaton associated with RWF is similar, although the subscriber does not set (or unset) an array in the same way. In fact it is set by the originator of the call.

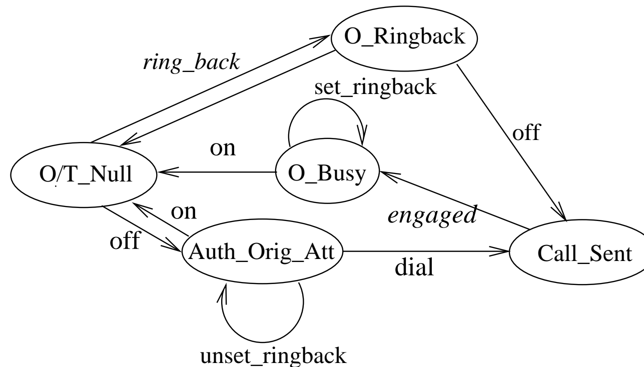

Figure3. Finite State Automaton for RBWF

8 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).
- New feature arrays are included, namely *CFU*, *CFB* etc. These are initialised within the `init` process according to which features are present within a given configuration. Note that the arrays associated with the *RBWF*, *RWF*, *OCO* and *TCO* features contain 0s and 1s only. As SPIN does not support bit arrays, these arrays are stored as arrays of bytes, using far more memory than necessary. Thus we could have saved valuable space by using bitvectors [52] rather than arrays in this case. However, for simplicity (in describing the *feature_lookup* below, for example) and consistency (with respect to the other features) we have used byte arrays.

- 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 dialed 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 either of the ringback features, RBWF or RWF, while straightforward, significantly increases the complexity of the underlying state-space. The reason for this is that an entirely new call state (*O_Ringback*) must be included in each case. In addition an array, *rgbknum* and *retnum* respectively, must be included to record the last process to whom a ringback has been initiated. 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 these arrays (c.f. discussion above about variable reinitialisation).
- To ensure that all variables are initialised, we again use 6 as a default value (assuming that the total number of users is 4). 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 "cancel ringback" in RBWF. We do not use an additional value for the latter, so as not to increase the size of the state-space.

8.1 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, as follows.

```
inline feature_lookup(q1,id1,st)
{
 do
 ::((id1!=7)&&(st==st_dial)&&(CFU[id1]!=6))
      -> id1=CFU[id1];q1=chan_name[id1]
 ::((id1!=7)&&(st==st_dial)&&(CFB[id1]!=6)&&(len(q1)>0))
      -> id1=CFB[id1];q1=chan_name[id1]
 ::((st==st_dial)&&(ODS[selfid]==id1))
      -> st=st_unobt
 ::((st==st_call2)&&(OCS[selfid]==id1))
      -> st=st_unobt
 ::((st==st_call2)&&(TCS[id1]==selfid))
      -> st=st_unobt
 ::((st==st_dial)&&(RBWF[selfid]==1)&&(id1==7)&&(rgbknum[selfid]!=6))
      -> rgbknum[selfid]=6;st=st_redial
 ::((st==st_idle)&&(RBWF[selfid]==1)&&(rgbknum[selfid]!=6)&&
    (len(chan_name[rgbknum[selfid]])==0))
      -> st=st_rback1
 ::((st==st_rback2)&&(RBWF[selfid]==1)&&(rgbknum[selfid]!=6))
      ->
      if
```

```
     ::dev=off;id1=rgbknum[selfid];q1=chan_name[id1];
       rgbknum[selfid]=6;st=st_call1
     ::/*fail to respond to ringback tone*/
       self?messchan,messbit; assert(messchan==self);
       messchan=null; messbit=0;rgbknum[selfid]=6;st=st_preidle
     fi
::((st==st_busy)&&(RBWF[selfid]==1)&&(rback_flag==0)&&(id1!=selfid))
     ->
     if
     ::rgbknum[selfid]=id1
     ::skip
     fi;
    rback_flag=1
::((st==st_idle)&&(RWF[selfid]==1)&&(retnum[selfid]!=6)&&
   (len(chan_name[retnum[selfid]])==0))
     -> st=st_rback1
::((st==st_rback2)&&(RWF[selfid]==1)&&(retnum[selfid]!=6))
     ->
     if
     ::dev=off;id1=retnum[selfid];q1=chan_name[id1];
       retnum[selfid]=6;st=st_call1
     ::/*fail to respond to ringback tone*/
       self?messchan,messbit; assert(messchan==self);
       messchan=null; messbit=0;retnum[selfid]=6;st=st_preidle
     fi
::((st==st_busy)&&(RWF[id1]==1)&&(ret_flag==0)&&(id1!=selfid))
     -> retnum[id1]=selfid; network_ev_action(ret_alert); ret_flag=1
::((st==st_call2)&&(OCO[id1]==1))
     -> st=st_unobt
::((st==st_idle)&&(TCO[selfid]==1))
     -> st=st_blocked
:: else
     ->break
od
}
```

The parameters $q1$, $id1$, and st take the values of the current partner, partnerid and state of a user when a call to the *feature_lookup* inline is made. Notice, for example, that st can take two values relating to the $Call_Sent$ state, st_call1 and st_call2. This is to differentiate between the state *to* which the process is to be directed on leaving the inline function and the state *from* which the process has entered the inline function respectively. As there is no redirection necessary from the O_Busy state when either of the ringback features are present, the $rback_flag$ and ret_flag variables have been introduced to prevent the relevant guards from remaining true after the corresponding transitions have taken place (that is, to prevent an infinite loop). Statements within *feature_lookup* pertaining to features that are not currently active are automatically commented out (see section 10.2).

feature_lookup encapsulates centralised intelligence about the state of calls i.e. it encapsulates what is known as single point call control. (*IN* supports both single and multiple point call control.) This greatly simplifies the detection of MU feature interactions. Although MU interactions can be detected in a distributed architecture, i.e. in a multiple point call control, the negotiation required between switches creates a huge and irrelevant (for our purposes) overhead. We therefore have single point call control.

We note that type I interaction detection is greatly facilitated by the form of *feature_lookup*: the majority of such interactions can be found by examining

the nondeterminism of the guards. For example, we can immediately see that the first two guards overlap if the state is *st_dial* and the process subscribes to both CFU and CFB.

8.2 Incorporating Features within the Promela Specification

In order to illustrate how combinations of features are incorporated into the specification, we use a specific example. Suppose that there are four users and two active features, namely CFU and TCS, where user 0 forwards to user 1 and user 2 screens calls from user 0.

Consider a specification of the basic call model (see section 3) to which the *feature_lookup* inline, feature arrays, the *rgbk_num* and *ret_num* arrays and the *O_Ringback* state have been added.

Firstly the *feature_lookup* inline is adapted so that all lines of code that do not correspond to either CFU or TCS are removed. Thus the *feature_lookup* inline becomes:

```
inline feature_lookup(q1,id1,st)
{
 do
 ::((id1!=7)&&(st==st_dial)&&(CFU[id1]!=6))
    ->id1=CFU[id1];q1=chan_name[id1]
 ::((st==st_call2)&&(TCS[id1]==selfid))->st=st_unobt
 ::else->break
od
 }
```

Secondly the *CFU* and *TCS* arrays are initialised thus:

```
CFU[0]=1;
CFU[1]=6;
CFU[2]=6;
CFU[3]=6;

TCS[0]=6;
TCS[1]=6;
TCS[2]=0;
TCS[3]=6;
```

All references to other feature arrays are removed. Finally the entire *O_Ringback* state is removed, together with the *rgbk_num* and *ret_num* arrays and all references thereof, as neither ringback feature is present.

In fact, for any combination of features, such changes are made automatically using a Perl script to create a specification from a template file (see section 10.2).

9 Temporal Properties for Features

The properties for features are more difficult to express than those for the basic service. In order to reflect accurately the behaviour of each feature, great attention must be paid to the *scope* of each property within the corresponding LTL formula, as described in section 5.3 (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 dialing party is in*

the Auth_Orig_Att 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 dialing party enters the *Auth_Orig_Att* state, and the second *after* the dialing party emerges from the *Auth_Orig_Att* state.

The properties are designed to reflect the expected behaviour of *our particular specification*. For example, we expect the forwarding features to be initiated *immediately* upon dialing. Thus we do not anticipate a user being able to hang up before a (redirected) connection is attempted. Property 7 reflects this expectation.

For the forwarding features, we want to capture the notion of a call attempt being made from user i to user j. We can not insist that a connection is achieved as the line may be busy. In previous work [4] it was deemed sufficient that $partner[i]$ is set to $chan_name[j]$. But, this does not completely capture the notion, with the consequence that we may miss some interactions (see section 10.4). Here we use the proposition below to represent the statement: *a call attempt is made from user i to user j*

$$\begin{aligned} & (partner[i] == chan_name[j]) \\ \&\& \ & ((User[proci]@O_Busy)||(User[proci]@O_Alerting)). \end{aligned}$$

We use the shorthand $att(i, j)$ to represent this proposition.

In some of the properties, propositions relate to the value of a variable ($dialed[i]$ say). It is often necessary to refer to the value of a variable *at a particular point in the service*. For example, in property 7 we are interested in cases when the value of the variable $dialed[i]$ is j when process i is at the *Call_Sent* state. (The value of $dialed[i]$ may be equal to j at a later state, *O_No_Answer* say, by which time it is too late to determine whether $process[i]$ subsequently attempts a call to the new partner.) Hence we enhance the proposition $dialed[i] == j$ with the extra condition that $(User[proci]@Call_Sent)$.

In section 5.2 we discuss the use of @ statements and event variables to track the progress of a process. The particular property dictates which of these to use. However, sometimes great caution must be exercised. For example, it is tempting to replace proposition $att(i, j)$ with the proposition $new_att(i, j)$ below:

$$\begin{aligned} & (partner[i] == chan_name[j]) \\ \&\& \ & ((network_event[i] == engaged)||(network_event[i] == oring)). \end{aligned}$$

However, $network_event[i]$ only takes the value *engaged* (or, at least, it is only visible to the *never-claim* as such) after $process[i]$ leaves the *O_Busy* state and proceeds to the *Preidle* state, by which time $partner[i]$ has been reset to the default value of *null*. Thus it is essential that an @ statement is used here. As far as possible the properties either use @ statements throughout or event variables. This is because the use of both would increase the number of variables that would be required, which would increase the size of the resulting state-space unnecessarily. Unfortunately, for the properties associated with the ringback features it is necessary to use both types of statement in order to fully express the desired behaviour.

Some of the properties below refer to the status of the device associated with a process. That is they refer to the device being *on* or *off*. In order to make this visible to the never-claim, the local variable *dev* must be replaced throughout the corresponding model by a global array of *dev* variables.

We have been especially careful to avoid the $\langle\rangle$ and X (next-time) operators in our properties. Both of these operators carry an increase in the cost of verification (due to the necessity of adding the *weak fairness* constraint and the prohibition of the use of partial order reduction respectively, see section 2.5). Sometimes the use of these operators is essential. However, we have managed to avoid their use in this instance, via judicious use of the U operator.

In each property 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 specification-generating script (see section 10.2).

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.

LTL: $[](p \rightarrow (r\mathcal{P}q))$

$p = ((dialed[i] == j)\&\&(User[proci]@Call_Sent))$, $r = att(i,k)$,
$q = (dev[i] == on)$.

Property 8(a) – 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.

LTL: $[](((u \wedge t) \wedge ((u \wedge t)U((\neg u) \wedge t \wedge p))) \rightarrow (r\mathcal{P}q))$

$p = ((dialed[i] == j)\&\&(User[proci]@Call_Sent))$, $t = (full(chan_name[j]))$, $r = att(i,k)$, $u = (User[proci]@Auth_Orig_Att)$,
$q = (dev[i] == on)$.

Property 8(b) – CFB Assume that user j forwards to k.
If user i rings user j when j is not busy then a connection between i and j will be attempted before user i hangs up.

LTL: $[](((u \wedge t) \wedge ((u \wedge t)U((\neg u) \wedge t \wedge p))) \rightarrow (r\mathcal{P}q))$

$p = ((dialed[i] == j)\&\&(User[proci]@Call_Sent))$, $t = (empty(chan_name[j]))$,
$r = att(i,j)$, $u = (User[proci]@Auth_Orig_Att)$, $q = (dev[i] == on)$.

Property 9 – OCS Assume that user i has user j on its screening list, $i \neq j$.

No connection from user i to user j is possible.

LTL: $[](\neg p)$

$p = (connect[i].to[j] == 1)$.

Property 10 – ODS Assume that user i has user j on its screening list, $i \neq j$.
User i may not dial user j.

LTL: $[](\neg p)$

$p = (dialed[i] == j)$.

Property 11 – TCS Assume that user i has user j on its screening list, $i \neq j$.
No connection from user j to user i is possible.

LTL: $[](\neg p)$

$p = (connect[j].to[i] == 1)$.

Property 12 – RBWF Assume that user i has RBWF.
If user i has requested a ringback to user j ($i \neq j$) (and not subsequently requested a ringback to another user) and subsequently user i is at O/T_Null when users i and j are both free, (and they are still free when user i is no longer at O/T_Null) then user i will hear the ringback tone.

LTL: $[]\neg((p\&\&q\&\&r\&\&s)\&\&((p\&\&q\&\&r\&\&s)U((p\&\&(\neg q)\&\&r)\&\&((\neg t)Uq))))$

$p = (rgbknum[i] == j)$, $s = (len(chan_name[i]) == 0)$,
$q = (User[proci]@O/T_Null)$, $r = (len(chan_name[j]) == 0)$,
$t = (network_event[i] == ringbackev)$.

Property 13 – OCO Assume that user j has OCO.
No connection from user i to user j is possible.

LTL: $[](\neg p)$

$p = (connect[i].to[j] == 1)$.

Property 14 – TCO Assume that user j has TCO.
No connection from user j to user i is possible.

LTL: $[](\neg p)$

$p = (connect[j].to[i] == 1)$.

Property 15(a) – **RWF** Assume that user j has RWF.
If user i calls user j when user j is busy ($i \neq j$), then user i will hear the ret_alert tone and retnum[i] will be set to j (before user i returns to the O/T_Null state).

LTL: $[]\neg((p\&\&q\&\&v)\&\&((p\&\&q\&\&v)U(p\&\&v\&\&(\neg q)))\&\&((\neg(r\&\&s))U(t)))$

$p = (len(chan_name[j]) > 0)$, $q = (User[proci]@O_Busy)$,
$v = (partner[i] == chan_name[j])$, $t = (User[proci]@O/T_Null)$,
$r = (retnum[j] == i)$,
$s = (network_event[i] == ret_alert)$.

Property 15(b) – **RWF** Assume that user j has RWF.
If user i has requested a ringback from user j ($i \neq j$) (and subsequently no other user has requested a ringback from user j), and user j is at O/T_Null and and users i and j are both free, then user j will hear the ringback tone.

LTL: $[]\neg((p\&\&q\&\&r\&\&s)\&\&((p\&\&q\&\&r\&\&s)U((p\&\&(\neg q)\&\&r)\&\&((\neg t)Uq))))$

$p = (retnum[j] == i)$, $s = (len(chan_name[i]) == 0)$,
$q = (User[procj]@O/T_Null)$, $r = (len(chan_name[j]) == 0)$,
$t = (network_event[j] == ringbackev)$.

10 Feature Interaction

In this section we consider only systems of 4 processes. Generalisation is discussed in section 11.

A property based approach to feature interaction detection assumes a formal model of the entire system and a given set of properties (usually temporal) associated with the features. In our case the formal model is the underlying Kripke structure associated with the Promela specification (see section 2.3).

Two features are said to *interact* if a property that holds for the system when only one of the features is present, does not hold when both features are present. For features f_1 and f_2 we define feature interaction as follows:

Definition 2. *Let $\mathcal{M}$ be the model of a system of N components in which no features are present and let $\mathcal{M}(f_1)$, $\mathcal{M}(f_2)$ and $\mathcal{M}(f_1 \cap f_2)$ be models in which only f_1, only f_2 and both f_1 and f_2 have been added respectively. If ϕ_1 and ϕ_2 are properties that define f_1 and f_2 respectively then f_1 and f_2 are said to interact if*

$$\mathcal{M}(f_1) \models \phi_1 \text{ but } \mathcal{M}(f_1 \cap f_2) \not\models \phi_1, \text{ or}$$
$$\mathcal{M}(f_2) \models \phi_2 \text{ but } \mathcal{M}(f_1 \cap f_2) \not\models \phi_2.$$

Note that this definition is relatively high-level, it does not contain details of the configuration. Thus it does not distinguish between SC (single component) and MC (multiple component) interactions. When we report on results later

(section 10.3) we will make this distinction. Note also that this analysis will only reveal interactions that exist with respect to the particular properties ϕ_1 and ϕ_2. For complete analysis it may be necessary to perform analysis for a suite of properties for each feature, or to conjoin properties.

10.1 Feature Validation

Before we can perform pairwise analysis of features, we must ensure that, for any feature, the model containing that feature in isolation satisfies the associated feature property. That is, using the notation above, we must show that for any feature f with associated property ϕ, $\mathcal{M}(f) \models \phi$.

Features and properties are labelled according to table 2. Each property must be checked for all relevant values of i and j (and k if appropriate).

Table2. Features and properties

Feature	Property ϕ
CFU	property 7
CFB	(property 8a) && (property 8b)
OCS	property 9
ODS	property 10
TCS	property 11
RBWF	property 12
OCO	property 13
TCO	property 14
RWF	(property 15a) && (property 15b)

Exploiting Symmetry A brute force approach quickly suffers a combinatorial explosion. For example, to verify property 7 (for all suitable networks of processes) with only four users, we need to check 48 cases (4 choices for i and j and 3 choices for k). We can eliminate the need to consider all cases by appealing to symmetry.

Consider the simplest case, where i, j, and k are distinct. Any permutation σ taking i, j and k to distinct i', j' and k' will preserve the transition relation of the underlying Kripke structure. The two models will be bisimilar and hence satisfy the same LTL formula, provided the formula is suitably renamed under σ. More formally, let $\mathcal{M}$ be a Kripke structure, s a state and ϕ an LTL formula involving only i, j, and k-indexed propositions, and σ a permutation with $\mathcal{M}_\sigma$, σs and $\sigma\phi$ the appropriate transformations under σ. There is a bisimulation between $\mathcal{M}$ and $\mathcal{M}_\sigma$ and so we have:

$$\mathcal{M}, s \models \phi \leftrightarrow \mathcal{M}_\sigma, \sigma s \models \sigma\phi.$$

The proof follows from the result that CTL* is adequate with respect to bisimulation [2]. This result relating permutations is similar, yet different to the result for symmetry groups and CTL* formulae [11]. The formulae, besides being in LTL (rather than CTL*), must be transformed under σ, unlike the symmetry group result where it may not be transformed and must be invariant under all permutations. This is because the symmetry group result is over the quotient structure, rather than individual (unquotiented) structures. We consider the individual structures and the permuted properties because our motivation is different: we require to reduce the number of cases to model check, not the state-space.

Application of this result allows us to exclude from consideration permutations of most of the relevant properties. For example, for property 7 we need consider only 3 cases ($i = 0, j = 0, k = 1$; $i = 1, j = 0, k = 1$; $i = 2, j = 0, k = 1$). All other cases are just a permutation of one of these.

Feature Validation Results For every feature f, associated property ϕ and choice of parameters, a Promela specification was constructed and verified. In each case the property was satisfied.

Clearly it is not possible to give detailed results for each relevant pair of features and properties. However in table 3 we give detailed verification results for each property for an example configuration of (distinct) parameters i, j (and k), when checked against a suitable network of processes.

Table3. Verification Results – feature properties

Property	Depth ($\times 10^6$)	States ($\times 10^5$)	Mem	Time	state-vector
7	0.6	3.0	12.6	15	116
8a	1.4	9.4	36.8	43	116
8b	1.4	7.9	32.7	40	116
9	1.2	6.1	23.8	29	124
10	1.1	6.4	25.4	30	136
11	1.1	6.3	25.1	29	136
12	11.2	59.5	263.7	385	132
13	0.6	3.1	13.0	16	136
14	0.5	3.2	12.9	16	136
15(a)	12.4	67	296.3	438	132
15(b)	12.3	70.2	308.7	476	132

10.2 Interaction Analysis

We now consider the case where two features are present. That is, we consider networks of 4 processes in which features f_1 and f_2 are present and determine

whether an interaction exists, according to definition 2. For any (distinct) pair of features f_1 and f_2 we need to determine whether the model $\mathcal{M}(f_1 \cap f_2)$ satisfies properties ϕ_1 and ϕ_2, associated with f_1 and f_2 respectively (see table 2). If not then if the two features are associated with the same process we have a SU interaction, otherwise we have a MU interaction.

Not that the analysis is *pairwise*, known as 2-way interaction analysis. While at first sight this may seem 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 [38]. A similar approach to dynamic analysis is taken, for example, by [48].

Automatic Specification Generation and Feature Interaction Before features were added to the basic call model, global variables could be "turned off" manually (i.e. commented out) or replaced by local variables when they are not needed for verification. The addition of features requires even more variables to be selectively turned on and off, or set to different values (see section 8.2).

This is both time-consuming and error prone. Therefore, we employ a Perl script to generate, for any combination of features and properties, a specification from a template file. Each generated specification also includes a header containing information about which features and properties have been chosen in that particular case, making it easier to monitor model checking runs.

The interaction analysis itself 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, to check for SU interaction in the CFB, ODS case alone, there are potentially 192 cases to investigate. Clearly we can again employ symmetry to reduce the number of cases. In the CFB, ODS case discussed above for example, it is only necessary to check scenarios in which $CFU[0] = 1$, $ODS[0] = m$, for all $m \neq 0$. Further use of symmetry is also possible. However, it is far more difficult to generalise when such further reduction can be made when two features are involved, than when just one feature is active (see section 10.1). For example, the amount of reduction that is possible depends on whether the two features considered are the same, whether we are looking for SU or MU interaction, and the particular property under consideration. For this reason, for most combinations of features, it is simpler to apply only the broad symmetry reduction described above. This enables the analysis to be fully automated (see below), without the need to consider each case individually. It is necessary, for cases where two ringback features are involved, to further reduce the number of cases using symmetry, as each verification takes approximately 90 minutes. However, we do not discuss this in detail here. The generalisation of symmetry reductions in the two feature case is the subject of further work.

Despite symmetry reduction, there are still a large number of cases to check. 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 specification/model,
– automatic SPIN verification of model and recording of feature interaction results.

Scenarios leading to feature interactions are recorded. A report of 1 error indicates an interaction. Once (if) an SU interaction is found, or, if no SU interactions exist, after it has been shown that no SU interaction exists, the search for MU interactions commences. If an MU 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: SU

/*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: MU
```

10.3 Feature Interaction Results

The tables in figure 4 give the interactions found (using automated specification generation and analysis) for pairs of features in both the SU case and the MU case. A $\surd$ in the row labelled by feature f means that the property ϕ associated with f is violated whereas a $\times$ indicates that no such violation has occurred. Two features f_i and f_j (that is, the features associated with rows i and j in the table) interact if and only if there is a $\surd$ in position (i, j) and/or a $\surd$ in position (j, i).

The tables are not symmetric. For example, there is an (ODS, CFU) MU interaction, but not a (CFU, ODS) MU interaction. To understand why, consider the witness scenario generated: an (ODS, CFU) MU interaction, under the assignment $ODS[0] = 1$, and $CFU[1] = 2$. Observe also the relevant guards in the inline *feature_lookup*. There are two choices, call them the "ODS choice" and the

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

(a) SU

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

(b) MU

Figure4. Feature Interaction Results - four users

"CFU choice". When *feature_lookup* takes the former, there is no interaction: both property 7 and property 10 are satisfied. One can understand this as ODS having precedence. However, there is a computation sequence where the latter choice is taken; in this case CFU has precedence and property 10 is violated because user 0 has *dialed* user 1 – before the call is forwarded to user 2 (although clearly property 7 is satisfied).

We note that often, understanding why and how a property is violated will give the designer strong hints as to how to resolve an interaction. In particular, a precedence is often implied. Precedence is just one technique for resolving interactions; we do not pursue it further here.

Altogether, 11 interactions are uncovered. These are interactions between: CFU and CFU, CFU and CFB, CFU and OCS, CFU and TCS, CFB and CFU, CFB and CFB, CFB and OCS, CFB and TCS, ODS and CFU, ODS and CFB, and TCO and RWF. It is clear that the majority arise from a forwarding type of feature.

10.4 Discussion

Type II interactions (see section 2.1) depend very much on the *properties* expressing user intentions. We believe our properties to be more detailed than most other analyses, but still they are not complete. In particular they do not state what should *not* happen (i.e. the frame problem).

The particular implementation detail of features also influences the interaction results. For example, had we chosen to implement the OCO feature from within the *Auth_Orig_Att* state (rather than in the *Call_Sent* state), different interactions would have exhibited. For example, there would be a (MU) interaction between CFU and OCO.

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. When analysis results are not symmetric, this strongly suggests a precedence between features.

11 Generalisation of Results

We have analysed networks of four user processes. However, is this sufficient for full pairwise interaction analysis? In this section we investigate the conditions under which we can extend our results to networks of arbitrary size.

11.1 The Parameterised Model Checking Problem

An obvious limitation of the model checking approach is that only finite-state models can be checked for correctness. Sometimes however we wish to prove correctness (or otherwise) of *families* of (finite-state) models. That is to show that, if $\mathcal{M}_N = \mathcal{M}(p_0||p_1||\ldots||p_{N-1})$ is the model of a specification consisting of N concurrent instantiations of a parameterised component p, then $\mathcal{M}_N \models \phi$ for all $N \geq n_0$, for some (small) value of n_0. In our case the p_i are the User processes, characterised by their process ids together with the list of features subscribed to by that user.

This is an example of the *Parameterised Model Checking Problem* which is, in general, undecidable [1]. The verification of parameterized networks is often accomplished via theorem proving [51], or by synthesising network invariants [9, 39, 56]. Both of these approaches require a large degree of ingenuity.

In some cases it is possible to identify subclasses of parameterised networks for which verification is decidable. Examples of the latter mainly consist of networks of N identical components communicating within a ring topology [19, 21] or networks consisting of a family of N identical *user* components together with a *control* component, communicating within a star topology [40, 21, 35]. A more general approach [18] considers a general parameterised network consisting of several different classes of components.

One of the limitations of both the network invariant approach and the subclass approach is that it can only be applied to networks in which each component (contained in the set of size N) is completely independent of the overall structure: adding an extra component (to this set) does not change the semantics of the existing components. A generalisation of data independence is used to verify arbitrary network topologies [12] by lifting results obtained for limited-branching networks to ones with arbitrary branching.

All of these methods fail when applied to asynchronously communicating components like ours, where components communicate asynchronously via shared variables.

We have developed an approach [6] based on abstraction and induction from which we can

1. validate the features for specifications consisting of any number of processes (and for which only one process has any features, namely the feature being validated), and
2. identify all pairs of features (f_1 and f_2 say) that do not interact, regardless of the numbers of processes, provided that f_1 and f_2 are the only features present.

We outline this approach in section 11.2 below.

11.2 The Abstraction Approach

For any feature f, we say that f is *indexed* by $I_f = \{i_1, \ldots i_r\}$ if the feature relates to $User[i_1], \ldots, User[i_r]$. For example if f is "$User[0]$ forwards calls to $User[3]$", then f is said to be indexed by 0 and 3. Similarly we say that a property ϕ is indexed by a the set I_ϕ where I_ϕ is the set of User ids associated with ϕ. For a (possibly empty) set of features $F = \{f_1, \ldots, f_s\}$ and property ϕ, we define the *complete index set* I of $\{\phi\} \cup F$, to be $I_{f_1} \cup \ldots \cup I_{f_s} \cup I_\phi$.

Suppose that we have a specification $p_0||p_1|| \ldots ||p_{N-1}$ of N telephone processes (with or without features) with associated model $\mathcal{M}_N$. For any $m < N$ we partition the User processes into two classes: $p_0 \ldots p_{m-1}$ are *concrete* and $p_m \ldots p_{N-1}$ are *abstracted*. We define a new process *Abstract*(m) which encapsulates the (externally) observable behaviour of the *abstracted* processes. Observable behaviour includes updates to global variables and communication channels, but excludes updates to local variables. Specifically, the process *Abstract*(m) has $id = m$ and an associated channel named $out_channel$. Since *Abstract*(m) encapsulates observable behaviour, we will use it to replace the behaviour of the abstracted processes. Call initiation from an abstracted process to a concrete process is replaced by a message of the form ($out_channel, 0$) from *Abstract*(m) to the relevant channel (*zero*, *one* etc.) which is always possible, provided the channel is empty. For example, the Promela proctype associated with the *Abstract*(m) process, when $m = 4$ is given below. Note that the value of the parameter *self* is passed via the command *run Abstract(out_channel)*.

```
proctype Abstract (chan self)
{do
:: zero!self,0
:: one!self,0
:: two!self,0
:: three!self,0
od}
```

Notice that *Abstract*(m) is very simple: it only includes behaviour for call initiation from an *abstracted* process to a *concrete* process (e.g. :: $zero!self, 0$ represents an *abstracted* process initiating a call to User 0). Any other message passing from the abstracted processes is now represented implicitly, by the non-deterministic choice available to the modified (*concrete*) processes, as described below.

To accommodate the fact that the concrete processes now communicate with *Abstract*(m) instead of an individual *abstracted* process, each *concrete* process $p_0 \ldots p_{m-1}$ is modified to p'_i, for $0 \leq i \leq m-1$, such that the *modified* processes behave exactly the same as the original (*concrete*) processes, except that, for $0 \leq i \leq m-1$:

1. process p'_i no longer writes to (the associated channels of) any of the processes $p_m, p_{m+1}, \ldots, p_{N-1}$ (the *abstracted* processes), instead there is a non-deterministic choice whenever such a write would have occurred as to whether

the associated channel is empty or full (thus, whether the write is enabled or not).

2. An initial call request from any *abstracted* process to p'_i now takes the form $out_channel!m, 0$ regardless of which abstracted process initiated the call. When such a message arrives on p'_i's channel, p'_i may read it. Henceforth p'_i no longer reads from (the associated channels of) any of the abstracted processes. Instead, p'_i makes a non-deterministic choice over the set of possible messages (if any) that could be present on such a channel.

Now we define a new abstract specification $abs(m)$ where

$$abs(m) = p'_0||p'_1|| \dots ||p'_{m-1}||Abstract(m).$$

We define $\mathcal{M}_{abs(m)}$ to be the associated model.

Suppose that in our specification of telephone processes the feature set and associated property ϕ have complete index set $\{0, 1, \dots, m-1\}$. We claim that if ϕ holds for the model associated with $abs(m)$ (namely $\mathcal{M}_{abs(m)}$), then it holds for $\mathcal{M}_N$. The abstraction is illustrated in figure 5, with the original specification on the left hand side and the specification $abs(m)$ appearing on the right hand side. Our main theorem holds because there is a simulation relation between the two specifications.

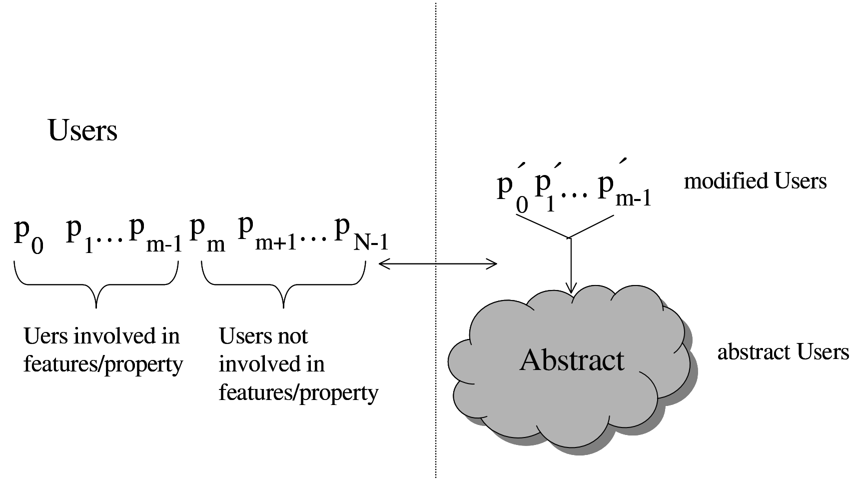

Figure5. Abstraction technique for an N-User telephone model

Theorem 1. *Let $\mathcal{M}_N = \mathcal{M}(p_0||p_1|| \dots ||p_{N-1})$ be a model of a specification of telephone processes in which only the features F are present, and ϕ a property.*

If the total index set of $F \cup \{\phi\}$ *is* $\{0, 1, \ldots, m-1\}$ *then* $\mathcal{M}_{abs(m)} \models \phi$ *implies that* $\mathcal{M}_N \models \phi$*.*

The full proof of Theorem 1, together with appropriate definitions, is given in Appendix 2. Here we provide a sketch of the proof. There are two steps. We may regard the first step as a data abstraction, and the second as behavioural abstraction.

The first step is to use data abstraction [10] to construct a (data) reduced model $\mathcal{M}_r^m$. The reduction involves abstracting the domains of all variables that can take the value of any process id to the set $\{0, 1, \ldots, m\}$. Each value $i < m$ is mapped to itself and all other values are mapped to m. The domains of all local variables of the abstracted processes and of all channels to and from the abstracted processes, are abstracted to the trivial set $\{true\}$. By data abstraction $\mathcal{M}_r^m \models \phi$ implies that $\mathcal{M}_N \models \phi$.

The second step establishes the fact that there is a simulation preorder [11] between $\mathcal{M}_r^m$ and $\mathcal{M}_{abs(m)}$, and thus properties are preserved.

Notice that $\mathcal{M}_{abs(m)}$ does not depend on N, i.e. for a fixed m, the abstracted model $\mathcal{M}_{abs(m)}$ is the same for a model of size N as it is for a model $\mathcal{M}_{N+1}$ in which an additional unfeatured process has been added. Thus our approach is similar to that of the invariant approach [9] but we have extended it to a featured paradigm.

Application of this theorem, for example, allows us to validate the CFU feature by model checking the abstract models corresponding to the three cases identified in section 10.1. (Note that we use symmetry to ensure that our total index set is always $\{0, 1, \ldots, m-1\}$ for some m.) Thus we check that

1. $\mathcal{M}_{abs(2)} \models \phi$, where ϕ is property 7, the feature is $CFU[0] = 1$, and the property parameters are $i = 0$, $j = 0$ and $k = 1$,
2. $\mathcal{M}_{abs(2)} \models \phi$, where ϕ is property 7, the feature is $CFU[0] = 1$, and the property parameters are $i = 1$, $j = 0$ and $k = 1$ and
3. $\mathcal{M}_{abs(3)} \models \phi$, where ϕ is property 7, the feature is $CFU[0] = 1$, and the property parameters are $i = 2$, $j = 0$ and $k = 1$.

In this way it is possible to validate all of the features using an abstract model $\mathcal{M}_{abs(m)}$ where $m \leq 3$ in all cases.

If there are two features, f_1 and f_2 say, in our feature set F then if ϕ is the property associated with one of the features, it follows that the complete index of $F \cup \phi$ is at most 4 when the features relate to the same user, and is at most 5 otherwise. For example, in order to show that a model with the features $CFU[0] = 1$ and $OCS[0] = 2$ satisfy property 7 with $i = 3$, $j = 0$ and $k = 1$ it is necessary to use an abstract model $\mathcal{M}_{abs(4)}$ but if the features are $CFU[0] = 1$ and $OCS[2] = 3$ and the parameters are $i = 4$, $j = 0$ and $k = 1$ it is necessary to use an abstract model $\mathcal{M}_{abs(5)}$. Note that, since $m \leq 5$ in all cases, and the id of the *Abstract* process is equal to m, it is still safe to use 6 as the default value of variables whose domains (otherwise) consist of the set of process ids.

11.3 Feature Interaction Results – any number of users

This method can only be used to find pairs of features that do not interact for any size of network (when they are the only features present). The only pairs of features that qualify therefore, are those which do not interact in the 4 user case. That is, pairs for which there is a × in the relevant position in the tables in figure 4.

Using our abstract model we are able to show that the results for the SU case for 4 users given in figure 4 hold for all number of user processes. However, in the MU case our results are less complete. In all cases it was possible to verify the abstract models for all values of m less than the maximum value. However, in some instances, for the maximum value of m the search depth became so big that the necessary search stack required too much memory for a full verification to be feasible. We give our results for any number of users in figure 6 below. Note that √ indicates that an interaction exists for some number of users, × that there is no interaction for any number of users, and • that it is not possible to verify the associated abstract model for all values of m.

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

(a) SU

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

(b) MU

Figure6. MU Feature Interaction Results - any number of users

We note that while these tables report both the presence and absence of interactions, we cannot use the abstraction method to detect interactions. This is because detection involves reasoning about *invalidity.* Specifically, the relations between the (original) models and the reduced and abstracted models are simulations (*not* bisimulations), thus invalidity of a formula for the abstracted model does not necessarily imply invalidity of the formula for the original model. However, due to the nature of our abstraction, for our set of features and properties, invalidity is often preserved.

12 Related Work and Comparison with Other Results

In this section we consider how our approach to feature interaction relates to other model checking approaches, and how our results compare. It is not straightforward to compare interaction analysis results, as even when researchers model

the (apparently) same features, actual implementation detail can profoundly affect the result. Nonetheless, we compare this work with other approaches, and with two sets of results.

12.1 Other Approaches

Model checking for feature interaction analysis has been investigated by others, notable approaches are those using COSPAN [20], Caesar [54], SMV [48], SPIN (the FeaVer project) [33, 32, 53] and a bespoke tool [36].

In [20], features and the basic service are described at only one level of abstraction, in a "sugared" form of LTL. The motivation for this single tiered approach (i.e. only logic, not logic and a form of state machine), is to exclude interactions which might be implementation dependent. It is difficult to compare results since we consider a different set of features (e.g. [20] does not include ring back features) and we deliberately chose to specify our system at a greater level of detail. In order to avoid state-space explosion, some "precision" of predicates is sacrificed in [20]. For example, they include the predicate $busy(x)$ (x is busy) rather than the more detailed $busy(x, y)$ (x is busy engaged in a call to y). Our specifications already include this level of detail (e.g. $partner[x] == chan_name[y]$), hence we have the ability to reason about more detailed behaviour. For example, we have included detail about the point in a call when features are initiated. The point of call can profoundly affect behaviour, especially the race conditions that may arise in an implementation. We can reflect this aspect by the nondeterminism in feature lookup. For example, in the case of forwarding, we explicitly assume the feature is initiated immediately upon dialling, but as we discussed earlier (section 10.4) if we change this assumption to another point in the call, the result would be a different set of interactions. We conjecture that the predicates of the COSPAN approach could be extended to the same level of detail; it would be interesting to see if state-explosion would be a problem, as we have found of number of aspects of SPIN optimisations and property refinements extremely useful. We note that our approach is motivated by our earlier work [54], where Caesar was used to check process algebra and μ-calculus. This too suffered from limitations imposed by state-explosion and the lack of (explicit) implementation detail afforded by state variables and asynchronous communication. Thus we adopted SPIN.

In [48], the authors present a similar two tiered approach using SMV (CTL and synchronous state machines). But similar to [20] the approach is less detailed, in order to make verification tractable, and because the SMV language is not very expressive. As a consequence, the authors note that they were forced to be more abstract than they wanted to be, for example, they restrict attention to two subscribers of the service with full functionality (plus two users with half functionality), due to state-space explosion problems. Call control is not independent, and because SMV is the underlying formalism, there is no explicit communication. Nevertheless, we regard this as a benchmark paper and we are able to demonstrate at least a similar set of properties and results; detailed comparison is given in section 12.2.

In [33, 32], the Promela model is extracted mechanically from call processing software code. A control-flow skeleton is created using a simple parser, and this skeleton is then populated with all message-passing operations from the original code. The motivation here is different from ours: theirs is a top down approach whereas ours is bottom up. We start with an abstract representation of a system and they with the full software code. Our approach is very much targeted at the design stage of system development, whereas theirs relies upon the existence of functioning code. It could be highly beneficial to apply the two techniques at both end of the development process. However, this would require access to the full call processing software code. As neither details of full pairwise interaction analysis nor of the model itself are provided, we cannot compare results.

In [53] a tool is described in which a subset of LTL properties can be specified using timelines via a simple graphical editor. The tool is useful when a large set of properties (with a similar structure) are to be verified. We have only a small set of properties, and we were primarily interested in their expression in LTL. Therefore we did not use this tool.

Motivation for the last paper [36] is detection and then resolution of interactions through predicate refinement, with respect to an underlying predicate taxonomy. Detection is by model-checking LTL formulae. The general approach and many of our interaction results are therefore similar, though we did not find that our properties fall into the same (two) suggested patterns. Model-checking details are not given, except to indicate that only given scenarios were explored. Thus we assume there was no complete pairwise analysis.

In summary, our approach is similar to the approach of Plath and Ryan [48]. Our main contribution is that in this case study we have presented a much more detailed specification, reflecting implementation detail such as explicit (and asynchronous) communication, yet verification is still tractable. Another contribution is generalisation, none of the above mentioned studies generalise results to more than three or four users.

12.2 Comparison with Other Results

The SMV study[48] We detected the same interactions between our common features CFU, CFB, RBWF, TCS, and OCS. In addition, we detected interactions between CFU and OCS, and between CFU and TCS. Plath and Ryan indicate that they missed these classical interactions, because aspects of SMV forced them to develop a model which was too abstract.

The FIW 2000 Contest[3] Another interesting benchmark for results is the feature interaction detection contest run in conjunction with FIW 2000 (*Feature Interaction Workshop 2000*). There is no definitive answer concerning which interactions *should* be detected, but results are given for analysis results from four research groups. Since three of our features: CFB, TCS and RBWF are included in the contest feature set, it is interesting to compare results.

CFB and TCS. Three out of four contestants detected this interaction, we also detect this interaction.

RBWF and TCS. All four contestants detected an interaction, but we did not. Why, is best illustrated by considering the following scenario. User i has RBWF and TCS with j on the screening list. i calls j, j is busy and i selects ringback. i will receive the ringback tone, when j is free (thus property 12 is satisfied). *But*, when i picks up the handset, they will hear the unobtainable tone (because the caller is on the screening list). Thus, strictly speaking, there is no interaction.

RBWF and CFB. Again, all four contestants detected an interaction, but we did not. Again, the reason is best illustrated by an example. User i has RBWF and j has CFB, i calls j, and j is busy. Calls are forwarded in the *Auth_Orig_Att* state, while ringback is set in the *Call_Sent* state. The call is forwarded immediately and no ringback is requested. Hence, there is no interaction.

These examples illustrate the importance of implementation details, in particular, at which point in the call a feature should be activated. This detail is often ambiguous in feature descriptions. We have made design choices which reflect our understanding of operational networks, and our decision to benchmark against the features given in [48]. Our results concerning the RBWF feature accord with the results of [48], but not with the FIW 2000 contest.

13 Conclusions

We have developed a Promela specification of a basic call service with nine features and performed pairwise feature interaction analysis on networks involving four users with full functionality. The analysis involves model checking with SPIN and is completely automated, making extensive use of Perl scripts to generate the SPIN runs, and symmetry to reduce case explosion.

The application area is a challenging one for model checking because formulating the right temporal properties for distributed systems is difficult, and the state-spaces for any realistic model quickly become intractable.

We have explained in some detail the difficulties of formulating temporal properties in LTL. The problems do not stem from the (lack of) path quantifiers, but rather from the complexity of the underlying distributed system.

We have demonstrated how a Promela specification can be optimised, without losing operational detail, by reinitialising variables in the appropriate way. Thus, we overcome classic state-explosion problems and our interaction analysis results are considerably more extensive than others (for example, [48]).

Since interactions are property violations, discovering these interactions by model checking always provides a counter-example. We thus gain a good understanding of why an interaction occurs, and this can help the redesign process.

When analysis results are not symmetric, this strongly suggests a precedence between features.

We have compared our results with others, and pointed out how some differences arise from implementation detail, in particular the points at which features are activated.

Finally, we have outlined a method for generalising results to networks of any size, when at most two features are present. The method is based on defining two further models, a reduced model and an abstract model, and simulation preorders between them. This allows us to infer general results about features which do not interact. We are currently investigating ways to extend our approach to the case where abstracted processes can have features.

Acknowledgements This work has been supported by a Daphne Jackson Fellowship, the EPSRC, and Microsoft Research. We would like to thank the anonymous referees for their valuable suggestions.

References

1. Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.
2. M. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
3. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), 2000.
4. M. Calder and A. Miller. Using SPIN for feature interaction analysis - a case study. In M.B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 143–162, Toronto, Canada, May 2001. Springer-Verlag.
5. Muffy Calder and Alice Miller. Analysing a basic call protocol using Promela/XSpin. In Gerard Holzmann, Elie Najm, and Ahmed Serhrouchni, editors, *Proceedings of the 4th Workshop on Automata Theoretic Verification with the SPIN Model Checker (SPIN '98)*, pages 169–181, Paris, France, November 1998.
6. Muffy Calder and Alice Miller. Automatic verification of any number of concurrent, communicating processes. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 227–230, Edinburgh, UK, September 2002. IEEE Computer Society Press.
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 L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 1–23. IOS Press (Amsterdam), May 1994.
8. Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mingardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety critical software with SPIN: an application to a railway interlocking system. In Langerak [42], pages 5–17.
9. E. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In Insup Lee and Scott A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR '95)*, volume 962 of *Lecture Notes in Computer Science*, pages 395–407, Philadelphia, PA., August 1995. Springer-Verlag.
10. E. Clarke, O. Grumberg, and D Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
11. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, Masachusetts, 1999.

12. S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, volume II, Las Vegas, Nevada, USA, June 2000. CSREA Press.
13. D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computer & Processors (ICCD'92)*, pages 522–525, Cambridge, Massachusetts, USA, October 1992. IEEE Computer Society.
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. Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In Olivier Danvy, editor, *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 105–118, San Antonio, Texas, January 1999. University of Aarhus. Technical report BRICS-NS-99-1.
16. E. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.
17. E. Emerson and J. Halpern. "sometimes" and "not never" revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.
18. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In David A. McAllester, editor, *Automated Deduction - Proceedings of the 17th International Conference on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254, Pittsburgh, PA, USA, June 2000. Springer-Verlag.
19. E. Emerson and K. Namjoshi. Reasoning about rings. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 85–94, San Francisco, California, January 1995. ACM Press.
20. A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In Calder and Magill [3], pages 179–192.
21. Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, July 1992.
22. R. Gerth, D. Peled, M. 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.
23. Orna Grumberg and David E. Long. Model checking and modular verification. In Jos C. M. Baeten and Jan Frisco Groote, editors, *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR '91)*, volume 527 of *Lecture Notes in Computer Science*, pages 250–265, Amsterdam, The Netherlands, August 1991. Springer-Verlag.
24. R.J. Hall. Feature combination and interaction detection via foreground/background models. In Kimbler and Bouma [37], pages 232–246.
25. 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):927–948, November 1998.
26. G. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25:981–1017, 1993.

27. G. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
28. G. Holzmann. State compression in Spin: Recursive indexing and compression training runs. In Langerak [42].
29. G. Holzmann. The engineering of a model checker: The Gnu i-protocol case study revisited. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of the 5th and 6th International Spin Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 232–244, Trento, Italy and Toulouse, France, 1999. Springer-Verlag.
30. G. Holzmann. *The SPIN model checker: primer and reference manual.* Addison Wesley, Boston, 2003.
31. G. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)*, volume 6 of *International Federation For Information Processing*, pages 197–211, Berne, Switzerland, October 1994. Chapman and Hall.
32. G. Holzmann and M. Smith. A practical method for the verification of event-driven software. In *Proceedings of the 21st international conference on on Software engineering (ICSE'99)*, pages 597–607, Los Angeles, California, USA, May 1999. ACM Press.
33. G. Holzmann and M. Smith. Software model checking - extracting verification models from source code. In J. Wu, S. Chanson, and Q. Gao, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)*, volume 156 of *International Federation For Information Processing*, pages 481–497, Beijing, China, October 1999. Kluwer.
34. *IN Distributed Functional Plane Architecture*, recommmendation q.1204, ITU-T edition, March 1992.
35. C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murϕ. *Formal Methods in System Design*, 14:273–310, 1999.
36. Bengt Jonsson, Tiziana Margaria, Gustaf Naeser, Jan Nystroem, and Bernhard. Steffen. Incremental requirement specification for evolving systems. In Calder and Magill [3], pages 145–162.
37. K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.
38. M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In Calder and Magill [3], pages 311–325.
39. R. P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proceedings of the eighth Annual ACM Symposium on Principles of Distrubuted Computing*, pages 239–247. ACM Press, 1989.
40. Robert P. Kurshan, M. Merritt, A. Orda, and S.R. Sachs. A structural linearization principle for processes. *Formal Methods in System Design*, 5(3):227–244, December 1994.
41. Leslie Lamport. What good is temporal logic? *Information Processing*, 83:657–668, 1983.
42. R. Langerak, editor. *Proceedings of the 3rd SPIN Workshop (SPIN'97)*, Twente University, The Netherlands, April 1997.
43. S. Leue and P. Ladkin. Implementing and verifying msc specifications using promela/Xspin. In J.-Ch. Gregoire, G.J. Holzmann, and D. Peled, editors, *Proceedings of the 2nd Workshop on the SPIN verification System*, volume 32 of *DIMACS*

Series in Discrete Mathematics and Theoretical Computer Science, pages 65–89, Rutgers University, New Jersey, USA, August 1996. American Mathematical Society.

44. Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical Report STAN-CS-90-1321, Stanford University, June 1990.
45. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1993.
46. D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996.
47. Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In D. Peled, V. Pratt, and G. 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*, pages 233–257, Princeton, New Jersey, USA, 1996. American Mathematical Society.
48. M. Plath and M. Ryan. Plug-and-play features. In Kimbler and Bouma [37], pages 150–164.
49. A. Pnuelli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
50. 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, Englewood Cliffs, NJ, 1994.
51. A. Roychoudhury and I.V. Ramakrishnan. Automated inductive verification of parameterized protocols. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer-aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 25–37, Paris, France, July 2001. Springer-Verlag. preliminary version of rora2.
52. Theo C. Ruys. Low-fat recipes for Spin. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th SPIN Workshop (SPIN 2000)*, volume 1885 of *Lecture Notes in Computer Science*, pages 287–321, Stanford, California, USA, September 2000. Springer-Verlag.
53. M. Smith, G. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Proceedings of the 5th IEEE International symposium on Requirements Engineering*, pages 14–22, Toronto, Canada, August 2001. IEEE Computer Society.
54. M. Thomas. Modelling and analysing user views of telecommunications services. In P. Dini, R. Boutaba, and L. Logrippo, editors, *Feature Interactions in Telecommunication Networks IV*, pages 168–182. IOS Press (Amsterdam), June 1997.
55. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In P. Alexander and P. Flener, editors, *Proceedings of the 15th IEEE Conference on Automated Software Engineering (ASE-2000)*, pages 3–12, Grenoble, France, September 2000. IEEE Computer Society Press.
56. Pierre Wolper and Vinciane Lovinfosse. Properties of large sets of processes with network invariants (extended abstract). In J. Sifakis, editor, *Proceedings of the International Workshop in Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, June 1989. Springer-Verlag.

Appendix 1: The Basic Service with features

We give here the optimised code for the basic service plus features. All code pertaining to feature behaviour (and to variables necessary only during property verification) is commented out.

```
mtype={on,off,dial,call,oring,tring,unobt,engaged,connected,disconnect,ringbackev,ret_alert,st_idle,
       st_blocked,st_unobt,st_rback1,st_rback2,st_dial,st_call1,st_call2,st_preidle,st_redial,st_busy};

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 */
chan partner[4];

/*byte CFU[4];*/ /*the ith member of these arrays switched to */
/*byte CFB[4]; */  /*default value of 6 if user[i]*/
/*byte OCS[4];*/  /*does not have this feature,*/
/*byte ODS[4];*/   /* and to identity  of user that */
/*byte TCS[4];*/   /*user[i] forwards to or screens otherwise*/
/*byte RBWF[4];*/ /*set to 0 or 1*/
/*byte OCO[4];*/ /*set to 0 or 1*/
/*byte TCO[4];*/ /*set to 0 or 1*/
/*byte RWF[4];*/ /*set to 0 or 1*/

/*mtype dev[4] = on;*/
/*byte dialed[4] = 6;*/
/*mtype network_event[4] = on;*/
/*mtype event[4] = on;*/
/*byte rgbknum[4] = 6; */
/*byte retnum[4] = 6; */

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

/*short p0=-1,p1=-1,p2=-1,p3=-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
 ::((id1!=7)&&(st==st_dial)&&(CFU[id1]!=6))
     ->id1=CFU[id1];q1=chan_name[id1]
 ::((id1!=7)&&(st==st_dial)&&(CFB[id1]!=6)&&(len(q1)>0))
     ->id1=CFB[id1];q1=chan_name[id1]
 ::((st==st_dial)&&(ODS[selfid]==id1))->st=st_unobt
 ::((st==st_call2)&&(OCS[selfid]==id1))->st=st_unobt
 ::((st==st_call2)&&(TCS[id1]==selfid))->st=st_unobt
 ::((st==st_dial)&&(RBWF[selfid]==1)&&(id1==7)&&(rgbknum[selfid]!=6))
     ->rgbknum[selfid]=6;st=st_redial
 ::((st==st_idle)&&(RBWF[selfid]==1)&&(rgbknum[selfid]!=6)&&
    (len(chan_name[rgbknum[selfid]])==0))
     ->st=st_rback1
 ::((st==st_rback2)&&(RBWF[selfid]==1)&&(rgbknum[selfid]!=6))->
```

```
    if
    ::dev=off;id1=rgbknum[selfid];q1=chan_name[id1];
      rgbknum[selfid]=6;st=st_call1
    ::/*fail to respond to ringback tone*/
      self?messchan,messbit; assert(messchan==self);
      messchan=null; messbit=0;rgbknum[selfid]=6;st=st_preidle
    fi
 ::((st==st_busy)&&(RBWF[selfid]==1)&&(rback_flag==0)&&(id1!=selfid))->
     if
     ::rgbknum[selfid]=id1
     ::skip
     fi;
     rback_flag=1
 ::((st==st_idle)&&(RWF[selfid]==1)&&(retnum[selfid]!=6)&&
    (len(chan_name[retnum[selfid]])==0))
     ->st=st_rback1
 ::((st==st_rback2)&&(RWF[selfid]==1)&&(retnum[selfid]!=6))->
    if
    ::dev=off;id1=retnum[selfid];q1=chan_name[id1];
      retnum[selfid]=6;st=st_call1
    ::/*fail to respond to ringback tone*/
      self?messchan,messbit; assert(messchan==self);
      messchan=null; messbit=0;retnum[selfid]=6;st=st_preidle
    fi
 ::((st==st_busy)&&(RWF[id1]==1)&&(ret_flag==0)&&(id1!=selfid))->
     retnum[id1]=selfid; network_ev_action(ret_alert);
     ret_flag=1
 ::((st==st_call2)&&(OCO[id1]==1))
    ->st=st_unobt
 ::((st==st_idle)&&(TCO[selfid]==1))
     ->st=st_blocked
 ::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;

O/T_Null:
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  O/T\_Null
       /*::state==st_rback1->
       self!self,0;state=on;goto ringback*/
    :: else->state=on
    fi;
    /*event_action(off);*/
    dev=off; self!self,0;goto Auth_Orig_Att
    /* 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 Present_Call
      :: 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};

Auth_Orig_Att:
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;
       /*dialed[selfid] = 0;*/
       partnerid=0
    :: partner[selfid] = one;
       /*dialed[selfid] = 1;*/
       partnerid=1
    :: partner[selfid] = two ;
       /*dialed[selfid] = 2;*/
       partnerid=2
    :: partner[selfid] = three;
       /*dialed[selfid] = 3;*/
       partnerid=3
    :: partnerid= 7;
 fi;
 state=st_dial;
 /* feature_lookup(partner[selfid],partnerid,state);*/
 if
 ::state==st_unobt-> state=on;partner[selfid]=null;partnerid=6;
   /*dialed[selfid]=6;*/
   goto O/T_Disconnect
 ::(state==st_dial&&partnerid!=7)-> state=on;goto Call_Sent
 ::(state==st_dial&&partnerid==7)->
   state=on;partner[selfid]=null;partnerid=6;
   /*dialed[selfid]=6;*/
   goto O/T\Disconnect
 ::(state==st_redial)->state=on;partnerid=6;
   /*dialed[selfid]=6;*/
```

```
   goto Auth_Orig_Att
 fi
 ::/*event_action(on);*/
   dev=on; self?messchan,messbit;assert(messchan==self);
   messchan=null;messbit=0;goto Preidle
   /*go onhook, without dialing */
 fi};


Call_Sent: /* check number called and process */
atomic
{/*event_action(call);*/
 assert(dev == off);
 assert(full(self));
 state=st_call2;
 /*feature_lookup(partner[selfid],partnerid,state);*/
 if
 ::state==st_unobt->state=on;partner[selfid]=null;partnerid=6;
   /*dialed[selfid]=6;*/
   goto O/T_Disconnect
 ::state==st_call2->state=on;skip
 fi;
 if
 :: partner[selfid] == self -> goto O_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 O_Alerting
       /* valid partner, write token to partner's channel*/
    :: full(partner[selfid]) -> goto O_Busy
       /* valid partner but engaged */
    fi
 fi};


O_Busy: /* number called is engaged, go onhook or trivial dial */
atomic
{assert(full(self));
 /*network_ev_action(engaged);*/
 if
 ::state=st_busy;
   /*rback_flag=0;*/
   /*ret_flag=0;*/
   /*feature_lookup(partner[selfid],partnerid,state);*/
   state=on;
   /*rback_flag=0; ret_flag=0;event_action (on);*/
   dev = on; self?messchan,messbit;assert(messchan==self);
   partner[selfid]=null;partnerid=6;messchan=null;
   /*dialed[selfid]=6;*/
   messbit=0; goto Preidle
   /*go onhook, cancel connection attempt */
/*:: event_action(dial); goto O_Busy*/ /* trivial dial */
fi};

 /*comment out entire ringback state when neither ring back
feature switched on */

/*ringback:
atomic
{state=st_rback2;
 feature_lookup(partner[selfid],partnerid,state);
 if
 ::state==st_call1->state=on;goto Call_Sent
 ::state==st_preidle->state=on;goto Preidle
 fi
};*/
```

```
O/T_Disconnect: /* number called is unobtainable, go onhook or trivial dial */
atomic
{assert(full(self));
 assert(partner[selfid]==null);assert(partnerid==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 O_Busy*/
   /* trivial dial */
 fi};


O_Alerting: /* 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 O_Active
   /* correct token */
 ::messbit==0->goto O_Alerting
   /* wrong token, not connected yet, try again */
 ::messbit==0->goto  O_No_Answer
   /* give up */
/*:: event_action(dial);messbit=0;goto O_Alerting*/
     /* trivial dial */
fi};


O_No_Answer: /*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;
 /*dialed[selfid]=6;*/
 messchan=null;messbit=0; goto Preidle;
 /* give up, go onhook */
};

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


O_Close: /* 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;
 /*dialed[selfid]=6;*/
```

```
 partnerid=6;messchan=null;messbit=0;
 goto Preidle};


Present_Call:
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 T_Alerting
 :: else->skip /* attempt has been cancelled */
 fi;
 /*network_ev_action(disconnect);*/
 self?messchan,messbit;
 partner[selfid]=null;partnerid=6; messchan=null;messbit=0;
 /*dialed[selfid]=6;*/
 goto Preidle
};


T_Alerting: /* 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 T_Active
    :: else ->/* wrong message, connection cancelled */
       /*network_ev_action(disconnect);*/
       self?messchan,messbit;
       /*event_action(on);*/
       dev=on; partner[selfid]=null;partnerid=6;messchan=null;messbit=0;
       /*dialed[selfid]=6;*/
       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;
    /*dialed[selfid]=6;*/
    messchan=null;messbit=0; goto Preidle
 fi
};


T_Active: /* 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 T_Active
 ::(messbit == 1 && dev == on) ->  /* trivial handset up */
   /*event_action(off);*/
   dev = off; messchan=null;messbit=0;goto T_Active
 ::(messbit == 0 && dev == on) -> /* connection is terminated */
```

```
   self?messchan,messbit;
   partner[selfid]=null;partnerid=6;messchan=null;messbit=0;
   /*dialed[selfid]=6;*/
   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;
   partnerid=6;messchan=null;messbit=0;
   /*dialed[selfid]=6;*/
   goto Preidle
 fi
};

Preidle:
atomic
{/*network_ev_action(on);*/
 /*event_action(on);*/
 goto  O/T\_Null
}

} /* 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 */

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

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

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

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

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

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

/*if user i has feature, set to 1, otherwise set to 0*/

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

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

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

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

Appendix 2: Proof of Theorem 1

We will assume throughout that the components $p_0, p_1, \ldots, p_{m-1}$ do not have any features other than those contained in the set F. The first stage of the proof of correctness of Theorem 1 involves the construction of a reduced model $\mathcal{M}_r^m$ via data abstraction [10] for any $m \leq N$. First we give some definitions:

Definition 3. *Let $X = \{x_0, x_1, \ldots, x_{l-1}\}$ denote a set of variables such that each variable x_i ranges over a set D_i. Then $D = D_0 \times D_1 \times \cdots \times D_{l-1}$ is called the domain of X. A set of abstract values $D' = D'_0 \times D'_1 \times \cdots \times D'_{l-1}$ is called an abstract domain of X if there exist surjections $h_0, h_1, h_2, \ldots, h_{l-1}$ such that $h_i : D_i \to D'_i$ for all $0 \leq i \leq l-1$. If such surjections exist they induce a surjection $h : D \to D'$ defined by*

$$h((x_0, x_1, \ldots, x_{l-1})) = (h_0(x_0), h_1(x_1), \ldots, h_{l-1}(x_{l-1})).$$

In the following definition (taken from [11]) data abstraction is used to define a reduced structure whose variables are defined over an abstract domain. See definition 1, section 2.3 for a definition of Kripke structure (assuming a single initial state s_0).

Definition 4. *Let $\mathcal{M} = (S, R, s_0, L)$ be a Kripke structure with set of atomic propositions AP and set of variables X with domain D. If D' is an abstract domain of X and h the corresponding surjection from D to D' then h determines a set of abstract atomic propositions AP'. Let $\mathcal{M}'$ denote the structure identical to $\mathcal{M}$ but with set of labels L' where L' labels each state with a set of abstract atomic propositions from AP'. The structure $\mathcal{M}'$ can be collapsed into a reduced structure $\mathcal{M}_r = (S_r, R_r, s_0^r, L'_r)$ where*

1. *$S_r = \{L'(s) | s \in S\}$, the set of abstract labels.*
2. *$s_0^r = L'(s_0)$.*
3. *$AP_r = AP'$.*
4. *As each s_r is a set of atomic propositions, $L_r(s_r) = s_r$.*
5. *$R_r(s_r, t_r)$ if and only if there exist s and t such that $s_r = L'(s)$, $t_r = L'(t)$, and $R(s, t)$.*

The following lemma (which is a restriction of a result proved in [10]) shows how we may use a reduced structure $\mathcal{M}_r$ to deduce properties of a structure $\mathcal{M}$.

Lemma 1. *If $\mathcal{M}$ and $\mathcal{M}_r$ are a Kripke structure and a reduced Kripke structure as defined in definition 4 then for any LTL property ϕ, $\mathcal{M}_r \models \phi$ implies that $\mathcal{M} \models \phi$.*

Our reduced model $\mathcal{M}_r^m$ is defined via the following abstract domains and corresponding surjections:

1. The domains of local variables of components $p_0, p_1, \ldots, p_{m-1}$ are unchanged. The abstract domains of local variables of components $p_m, p_{m+1}, \ldots, p_{N-1}$ are the trivial set $\{true\}$, and the associated surjections g, where $g(x) = true$ for all x in the original domain.

2. In $\mathcal{M}_N$ all other variables, apart from those associated with channel names or contents, have domains equal to the set $\{0, 1, \ldots, N-1\}$ (the set of component ids). Each of these variables have abstract domains equal to the set $\{0, 1, \ldots, m-1\}$ and a surjection from the original domain D to the abstract domain D' is given by $h_1 : D \rightarrow D'$ where

$$h_1(x) = \begin{cases} x & \text{if } x < m, \\ m & \text{otherwise} \end{cases}$$

for all $x \in D$.
3. In $\mathcal{M}_N$, the domains of channel variables such as *self* and *partner*, consist of the set of channel names $name[0], name[1], \ldots, name[N-1]$ (where $name[0]$, $name[1]$, etc. represent the channel names *zero*, *one*, etc.). The abstract domains for such variables is $name[0], name[1], \ldots, name[m]$ and the surjection h_2 is an obvious extension of h_1 above.
4. Similarly abstract domains for channel contents (a channel name and a status bit in each case) can be defined, and a surjection given in each case. The abstract domains for the variables of contents of channels associated with components $m, m+1, \ldots, N-1$ are the trivial set.

From Lemma 1 it follows that for any LTL property ϕ, $\mathcal{M}_r^m \models \phi$ implies that $\mathcal{M}_N \models \phi$.

The next stage of our proof involves showing that, for all $m \leq N$, $\mathcal{M}_{abs(m)}$ *simulates* $\mathcal{M}_r^m$. Again we provide some useful definitions:

Definition 5. *Given two structures $\mathcal{M}$ and $\mathcal{M}'$ with $AP \supseteq AP'$, a relation $H \subseteq S \times S'$ is a simulation relation between $\mathcal{M}$ and $\mathcal{M}'$ if and only if for all s and s', if $H(s, s')$ then*

1. $L(s) \cap AP' = L'(s')$
2. *For every state s_1 such that $R(s, s_1)$, there is a state s_1' with the property that $R'(s', s_1')$ and $H(s_1, s_1')$.*

If $H(s_0, s_0')$ we say that $\mathcal{M}'$ simulates $\mathcal{M}$ and denote this by $M \preceq M'$.

Lemma 2. *Suppose that $\mathcal{M} \preceq \mathcal{M}'$. Then for every LTL formula ϕ with atomic propositions in AP', $M' \models \phi$ implies $M \models \phi$.*

To prove that, for all $m \leq N$, $\mathcal{M}_{abs(m)}$ simulates $\mathcal{M}_r^m$, it is first necessary, for all $m \leq N$, to define a relation between the set of states of $\mathcal{M}_r^m$ (S_r^m say) and the set of states of $\mathcal{M}_{abs(m)}$ (S_{abs}^m say). We note that there is only one local state associated with the abstract component *Abs* (see section 11.2), and so we can ignore this aspect of (elements of) S_{abs}^m.

Suppose V is the set of variables associated with $\mathcal{M}_N$ and V_r a reduced set of variables, such that V_r is identical to V except that the local and global variables associated with components $p_m, p_{m+1}, \ldots, p_{N-1}$ have been removed. The atomic propositions relating to $\mathcal{M}_N$ is the set $AP = \{x = y : x \in V \text{ and } y \in D(x)\}$, where $D(x)$ is the domain of x. Let us consider the alternative set of atomic

propositions $AP' = \{x = y : x \in V_r \text{ and } y \in D'(x)\}$, where $D'(x)$ is the abstract domain of x. If we let L_r denote the labelling function associated with AP', then we can define a relation H between S_r^m and S_{abs}^m as follows: For $s \in S_r^m$ and $s' \in S_{abs}^m$, $H(s, s')$ if and only if $L_r(s) = L_r(s')$.

To show that H is a simulation relation, it is necessary to show that for all $(s, s') \in H$, every transition from (s, s_1) in $\mathcal{M}_r^m$ is matched by a corresponding transition (s', s_1') in $\mathcal{M}_{abs(m)}$, where $(s_1, s_1') \in H$.

Every transition in $\mathcal{M}_r^m$ either only involves a change to the global variables or involves a change to the value of the local variables of one of the (concrete) components. If the former is true, then the transition involves an initial message being placed on the channel of one of the (concrete) components $p_0, p_1, \ldots, p_{m-1}$ by one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$. This transition is reflected in $\mathcal{M}_{abs(m)}$ by a transition involving the *Abstract* component in which a message is placed on the channel of the concrete component. If t is a transition in $\mathcal{M}_r^m$ involving concrete component p_i then either t does not involve any component other than p_i or t involves only component p_i plus another concrete component or one or more of the following holds:

1. Component p_i is not currently in communication with another component and t involves a read from the channel of p_i as a result of the initiation of communication by one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$, or
2. component p_i is currently in communication with one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and t involves p_i reading a message from this component, or
3. component p_i is currently in communication with one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and t involves a call to the *feature_lookup* function.

(Notice that a write to one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ does not involve a change of state, as the abstract domains associated with the channel contents of each of these components is trivial.)

Let $t = (s, s_1)$ and suppose that $H(s, s')$ for $s' \in S_{abs}^m$. If $t = (s, s_1)$ does not involve any component other than p_i, or t involves p_i and another concrete component, there is clearly an identical transition $(s', s_1') \in \mathcal{M}_{abs(m)}$ such that $H(s, s')$.

If t involves a read from p_i's channel of an initial message sent by one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ then t is reflected by a transition in $\mathcal{M}_{abs(m)}$ (p_i will still read such a message). However, if t involves any other read from one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$, non-deterministic choice in p_i' (the corresponding component in $abs(m)$) ensures that an equivalent transition in $\mathcal{M}_{abs(m)}$ exists.

Suppose that p_i is currently involved in communication with one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and t involves a call from p_i to the *feature_lookup* function. As components $p_m, p_{m+1}, \ldots, p_{N-1}$ have no associated features, any guard g within the *feature_lookup* function that holds for a state in $\mathcal{M}_N$ (from which *feature_lookup* is called) will hold at the associated state in $\mathcal{M}_r^m$. If s is such a state and s' a state in $\mathcal{M}_{abs(m)}$ such that $H(s, s')$ then g holds at s' and it is clear that any transition t in $\mathcal{M}_r^m$ from s is reflected in $\mathcal{M}_{abs(m)}$.

Since clearly $H(s_0, s'_0)$, where s_0 and s'_0 are the initial states of $\mathcal{M}_r^m$ and $\mathcal{M}_{abs(m)}$ respectively, $\mathcal{M}_{abs(m)}$ simulates $\mathcal{M}_r^m$. From Lemma 2 we can conclude that, for any LTL property ϕ, if $\mathcal{M}_{abs(m)} \models \phi$ then $\mathcal{M}_r^m \models \phi$. Theorem 1 follows from Lemma 1.