Modelling a Telephone Switching System for Verification

Stephan Reiff and Muffy Calder

e-mail: {sreiff,muffy}@dcs.gla.ac.uk

University of Glasgow, Department of Computer Science, Glasgow, G12 8QQ, Scotland

Abstract

After identifying the general structure of a POTS (Plain Old Telephone System) Switching System we model individual components as communicating finite state machines. Formulating the finite state machines in PROMELA (Process Meta Language) allows us to verify general and specific properties of the model using the model-checker SPIN. Modelling operational systems imposes design decisions, which we will consider in the appropriate context.

1 Introduction

The development of telephone switching systems from manual switchboards to automated systems has required the development of control software. The development process of the switching software has in general been complex and evolutionary; this is sometimes reflected in the software which may be large, poorly structured and/or undocumented legacy code. Increasing user demands and advancing technology are forcing the provision of extensions to the software, usually in the form of features. Implementing features is not straightforward, as in addition to the desired interactions with the legacy system and other features, undesirable ones also occur. This is commonly known as the *feature interaction problem*. A typical example of an undesirable interaction is an inconsistency between *call waiting* and *call forwarding on busy*. The former requires to provide a call waiting tone, whereas the latter tries to forward the call – both are triggered by an incoming call on a busy line.

To improve the provision of features in telephone switching systems, we are investigating methods to detect and resolve feature interactions, within the context of legacy and/or proprietary code. This means that we cannot assume access to the (specification of the) internal behaviour of the switch control software, it may be that at best we can observe messages sent to/from other system components, and/or force a commit or roll-back of behaviour through a transactional interface ([CMM99]).

We are convinced that model-checking will provide results that can be used to improve operational and legacy systems as well as their development process. But to achieve

this, the model needs to be an appropriate abstraction of the operational system, i.e. realistic enough to represent the system while being simple enough to be formally verified.

We propose such a model in Section 2 and give details of the components in the subsequent section. In Section 4 we comment on the implementation in PROMELA (PROcess MEta LAnguage) – the input language for the model-checker SPIN([Hol91]). Sections 5 and 6 discuss verification issues, including both general and specific properties. Section 7 concentrates on related and further work to give the broader context of this work and finally we conclude by summarising our achievements so far.

2 A POTS Switching System

A high level view of a telephone switching system consists of three principal categories of components: the users, the operator and the network (Figure 1).

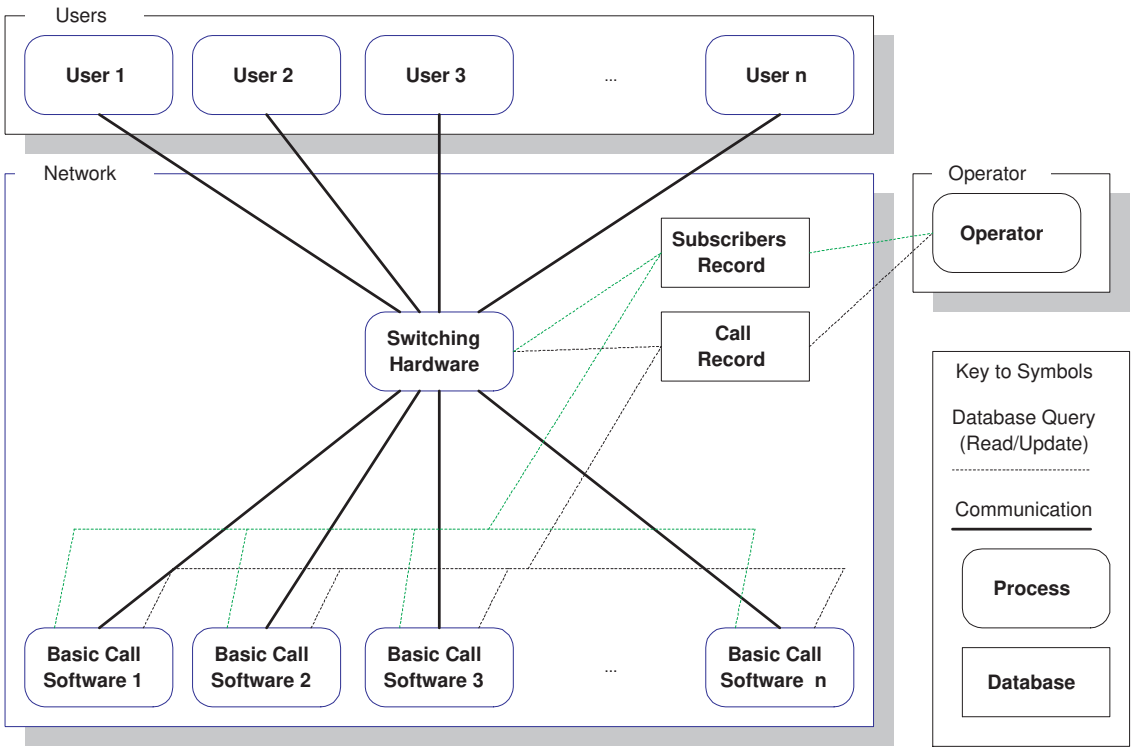

Figure 1: Overview of the System

The term 'user' is used to describe points of interaction between the network and the outside world, such as terminal devices, trunks (connections to other networks) or any other communication equipment. For simplification we only consider simple terminal devices (i.e. telephones) and assume that there is only one network.

The operator interactions are mainly database updates and enquiries (mostly for billing purposes) and do not concern us here.

The network consists of database components, basic call software processes and the switching hardware process. The database components contain the call records and the subscriber records, which are used for control and billing purposes. For example, we may query information about the status of a telephone line and information about user permissions from the subscribers record. Each user is assigned a basic call software process, which is only instantiated on demand and provides POTS (Plain Old Telephone System) call behaviour. The switching hardware process controls the communication (i.e. message transfer) between users and basic call software processes (and combinations thereof).

Although this model seems rather more simplistic than a PSTN (Public Switched Telephone Network) it does, for example, describe a PBX (Private Branch Exchange).

3 The Components of the Network

The process components are modelled as non-deterministic finite state machines (FSM's). We can compose FSM's, allowing communication via their respective input and output messages. When FSM's are composed to build the system, each is embedded within an environment formed by the others and the database components. We refer to this as the network, which itself is embedded in the larger environment as shown in Figure 1.

Note that the network environment resolves any non-determinism in its components. To achieve this, especially when there is a choice between reading an input message or another behaviour, we have adapted one of the fundamental ideas of Input/Output Automata ([LT88]): an input message is never blocked, but each FSM can decide when to send output messages.

Switching Hardware Process

The switching hardware is the main control process, i.e. a reactive process which scans input lines for incoming signals and transmits appropriate outgoing signals. Our model incorporates this scanning as sequential polling. Apart from being realistic, this also provides a fair mechanism. Therefore, the general behaviour of the switching hardware process can be described as:

```
loop
  for each input channel
    scan for message
    if message exists
```

```
      read message
      process message
    fi
  end for
end loop
```

Processing a message basically consists of identifying the destination and sending it on the right output channel. There are three cases:

1. a message from a user has to be sent on to the associated basic call software,
2. a message from a basic call software has to be sent to the associated user, or
3. a message from a basic call software has to be sent to another basic call software.

In our model this is the level of granularity of concurrency, that is the reading, processing and sending on of a message is an atomic action, i.e. it cannot be interrupted and occurs as one step from the viewpoint of the environment.

Basic Call Software Process

The behaviour of the basic call software process is a little more complex. Figure 2 shows the FSM describing the basic call software, with input messages in italics; all other messages are output messages. A message consists of two parts, an event and a numeric argument. The argument is usually an user identifier, or 0 when no argument is required. Some arrows are labelled with two messages separated by a semicolon, we shall regard this as a sequential composition of messages.

To illustrate the behaviour, consider an example of call set-up and clear-down to another user, say user 4 (i.e. consider originating behaviour only). Assume that the basic call software is in the `idle` state and user 3 is able to engage in the call.

Example 1 *The basic call software receives an (offhook, 0) message to which it responds with (dial_tone, 0). Next, a (dial, 4) messages triggers the sending of (ring_tone, 0) and (o_alert_request, 4). Assume user 4 goes offhook which leads to an (i_connect_request, 4) message being received, triggering (connect, 0) being sent. Now the two users are connected. The basic call software receives an (onhook, 0) message indicating that the connection is to be cleared down, triggering an (o_disconnect_request, 4) message being sent. At this point the basic call software is in its initial* `idle` *state and a new call can be made.*

Note that in the current BT (British Telecom) network, only the originator of a call can clear down the call (with exceptions like 999). However, many PBXs show symmetric behaviour, as modelled here.

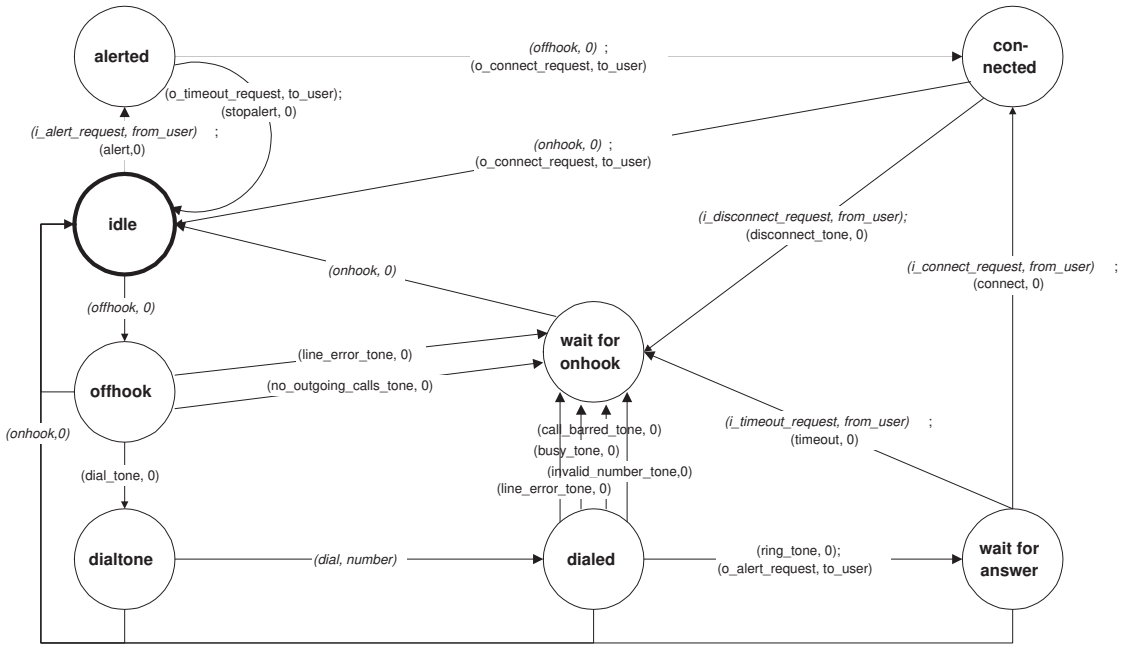

Figure 2: The Basic Call Software Process

4 Promela Implementation

The complete implementation of a two user model (consisting of two user processes, two basic call software processes and a switching hardware process) comes to more than 700 lines of PROMELA code. We do not include it here, but instead highlight some aspects.

Converting the FSM's into PROMELA is relatively straight forward, but nevertheless requires certain design decisions to be made.

Global Variables. We have decided to make the channels global, in order to reason about channels (c.f. Section 6). The only other global variables are an array of records representing the database components.

User processes. For the user process we made several design decisions, that are mostly justified by the behaviour of operational systems. A major problem involved synchronisation of messages and responses across the network. For example, consider the scenario that a user could go offhook followed quickly by an onhook, with a pending and wrongly delivered dial-tone generated as response to the offhook. Discussing the scenario with one of our industrial partners revealed that this is really what happens (at least in their system): the dial-tone is delivered. Hence, we made the decision that a user process reads any input message and discards it if it is irrelevant in the current state, thus being consistent with the requirement to process input messages before executing other behaviour. Further, a user can only send a new message when the previous one has been processed, an offhook message will not be delivered until the users status is free.

Basic Call Software. On the other hand, within the basic call software process we assert that only relevant messages are received. A PROMELA assertion statement is added for any non-valid message and any violation would obviously be flagged in a verification (c.f. Section 5). The basic call software process follows the FSM closely by representing the states with labelled sections of code.

Each such state section is encapsulated as an atomic step, which itself consists of an assumption about the users line-status and a multiple branching statement. The choice of the branch taken is dependent on the existence of a value on the input channel (recall that we force the processing of available input messages). Within each branch, we then have further assumptions and the requisite instructions for the current state.

It might seem unfair (in a technical sense) to have such high level atomic steps, but apart from cutting down the state-space, this is justified by our aims: we are basically modelling black-box components. Thus, it seems appropriate to model component state changes as atomic.

To illustrate the behaviour we give below two states of the basic call software process: the `idle` and the `dialled` state.

Initially the process is in the `idle` state, where it can either receive an (*offhook*, 0) or (*i_alert_request, from_user*) message, followed by a transition to another state after changing the line status and (in the latter case) sending an (*alert*, 0) message to the user. There is a further possibility: if a (*i_connect_request, from_user*) or (*i_timeout_request, from_user*) is received, they are out-of-sync and are ignored (i.e. they are read and the process remains in `idle`).

```
BC_IDLE:
        atomic{
        if
        :: sin[id]?m ->
               assert(cosr[id].t.ls == Free);
               assert(m.ev != i_disconnect_request);
               assert(m.ev != onhook);
               assert(m.ev != dial);
               if
               :: m.ev == offhook ->
                    cosr[id].t.ls = Busy;
                    goto BC_OFFHOOK;
               :: m.ev == i_alert_request ->
                    cosr[id].t.ls = Busy;
                    partner = m.param;
                    bcs_send_msg(alert, 0, sout[id]);
                    goto BC_ALERTED;
               :: (m.ev == i_connect_request) || (m.ev == i_timeout_request) ->
                    goto BC_IDLE;
               fi;
        fi;}
```

Now consider the `dialled` state; the state of the basic call process after a (*dial, number*) message has been received. In this state, the subsequent behaviour depends on the presence (or absence) of an input message. In the former case, the message is either the user has gone onhook, or it is an out-of-sync message from another basic call process. In the latter case, the class of service record is enquired for line status and user permissions, the dialled number is tested for validity and the appropriate notification is sent if necessary; otherwise the user is notified that the partner's phone is ringing.

```
BC_DIALLED:
           atomic{
           if
           :: full(sin[id]) ->
                    sin[id]?m;
                    assert(cosr[id].t.ls == Busy);
                    assert(m.ev != dial);
                    assert(m.ev != offhook);
                    assert(m.ev != i_disconnect_request);
                    assert(m.ev != i_connect_request);
                    if
                    :: m.ev == onhook ->
                         cosr[id].t.ls = Free;
                         clear_out(sout[id]);
                         goto BC_IDLE;
                    :: m.ev == i_timeout_request ->
                          goto BC_DIALLED;
                    :: (m.ev == i_alert_request) ->
                          bcs_send_msg(o_timeout_request, m.param, sout[id]);
                          goto BC_DIALLED;
                    fi;
           :: nfull(sin[id]) ->
                  assert(cosr[id].t.ls == Busy);
                  if
                  :: cosr[partner].t.ls == Busy ->
                       bcs_send_msg(busy_tone, 0, sout[id]);
                       goto BC_WAITFORONHOOK
                  :: cosr[id].s.ob == Long ->
                       bcs_send_msg(call_barred_tone, 0, sout[id]);
                       goto BC_WAITFORONHOOK
                  :: cosr[partner].t.ls == OutOfService ->
                       bcs_send_msg(line_error_tone, 0, sout[id]);
                       goto BC_WAITFORONHOOK
                  :: partner >= USERS ->
                       bcs_send_msg(invalid_number_tone, 0, sout[id]);
                       goto BC_WAITFORONHOOK
                  :: else ->
                       bcs_send_msg(ring_tone, 0, sout[id]);
                       /* no explicit state for bc_ringing_tone */
                       bcs_send_msg(o_alert_request, partner, sout[id]);
```

```
                goto BC_WAITFORANSWER
            fi;
        fi;}
```

5 Model-Checking: General Properties

The modelled system should obey certain general properties, for example absence of deadlocks and live-locks, and consistency with respect to in-line assertions. As SPIN automatically verifies these properties, it is only worth considering the form of our assertions in more detail.

Assertions can be based on values on channels or on values contained in the database components. Note that initially we introduced more assertions, but found during the simulation and verification that many of these were violated. Introducing new (or extending existing) guards to capture those cases, we remove the assertions but still retain the intended behaviour.

The assertions are invariants concerned with messages and the line status (which is stored in the subscriber records).

As an example, consider how to capture the following invariant using assertions. A basic call software process, in the `dialled` state, cannot receive (*dial*, *number*) or (*offhook*, 0) messages, as the associated user should not be able to send these. Further, (*disconnect*, *from user*) or (*connect*, *to user*) messages are also impossible: a precondition for a disconnect message is that the user must be connected, and right after dialling this is certainly not the case. A connect message is sent from another process and as such could occur, but another user should only have been able to send this after successful dialling (which in turn requires the dialled user to be free).

From the example, we can see that formulating the assertions requires an understanding of the global behaviour together with an understanding of the timing issues in the system; thus it is not trivial.

Assertions about the line status are easier; the line status is toggled when the basic call software receives an (*onhook*, 0) or an (*offhook*, 0) message. In the former case, the line status should be set to free, in the latter to busy. Assertions are made in the basic call software to ensure that the line status was indeed updated accordingly.

6 Model-Checking: Specific Properties

In addition to general properties we also want to check for specific behaviour. Examples of specific behaviour are:

- If user A calls user A, then user A should perceive a busy-tone.

- If there are no faults with the lines, user A is allowed to originate calls and user B is not busy, then eventually they can be connected.

Obviously it is not to hard to come up with further properties, illustrating expected or unexpected behaviour. A key point to note is that the properties should be based on observable behaviour in the system. For example a property such as *if we press the clear button on a calculator, the display memory buffer is cleared* cannot be expressed if we cannot observe the value of the buffer. This leads to the important question *What behaviour can we observe?*

Recalling that we want to reason about black-box systems, i.e. proprietary or legacy code, we cannot observe any internal behaviour. This excludes reasoning about call states in the model (i.e. `idle`, `dialling`, etc.). So, we have to consider other possibilities. We can observe values that are passed along channels (just as we can "spy on" (and actually write to as well) communication channels in an operational system [1]). With this assumption we base our reasoning exclusively on channels and the values passed along them.

First, consider how to formulate these properties (expressed in English) in a formalism we can use for verification, in this case LTL (linear temporal logic). Here we consider only the first of the above properties, namely *If user A calls user A, then user A should perceive a busy-tone (We assume that A has directory number 1).*

Assume the following propositions:

$p =$ *User A dials 1*

$q =$ *User A perceives a busy-tone*

Then the property is formalised in LTL by:

$$\Box(p \rightarrow \Diamond q) \tag{1}$$

But, if we translate the formula back to English (*it is always the case that when user 1 dials 1, user 1 eventually gets a busy-tone*) we realise it does not capture exactly the property we had in mind. Namely, we want to express that the busy-tone occurs as the *next* event. A naive approach is to propose

$$\Box(p \rightarrow \bigcirc q) \tag{2}$$

which seems to capture the intention. But, as we are dealing with a concurrent system, the next event observed by the user is not necessarily the next event that globally

[1] For experimental and prototyping purposes, we have access to a switching system from Mitel Corporation

happened, so this formula still does not capture our intention. The only solution is to state explicitly that *until* the user receives the busy-tone the user does not exhibit any other behaviour.

Assuming the additional propositions:

s = *User A does not send any other message*

t = *User A does not receive any message*

we propose that the following formula more accurately captures our intended property:

$$\Box(p \rightarrow ((s \wedge t)Uq) \tag{3}$$

We note, for verification of this kind it would be helpful to have certain new 'relativised' operators. For example, we would like some form of *local temporal operator* such as $\bigcirc_{UserA}$, with the obvious meaning *next for user A*. However, we do not pursue this further here.

Second, having clarified how to express the property in LTL, we need to make it available for SPIN in order to run an automatic verification. This process involves two steps: defining the propositions and generating the never claim. The latter is trivial because SPIN's claim generator automatically produces the never claim from the formula (3). The former, defining the symbols p, q, s and t, remains to be done.

For s and t the solution is simple, we just use the function `len(channel)` to determine whether the channel is empty:

```
#define s len(uout[1]) == 0
#define t len(uin[1]) == 0
```

Now consider p and q; guidance on how to reason about values currently on channels is given in ([Hol93], p. 1002), where the example shows that we can test with `channelname?[value]` for the existence of a certain constant value on the channel. But now, a further problem occurs, namely that this does not apply to structured messages (at least not if the structure is defined using `typedef`), which we are using. The channels are of the type `tmsg`, which is defined as:

```
typedef tmsg {mtype ev;                    /* Messagetype */
              int   param};
```

Clearly we would need `value` in `channelname?[value]` to be a constant of type `tmsg`. It seems that the only way to obtain the two constants we require (for p and q) is to define 2 global variables, `msg0` and `msg1` of type `tmsg` and initialise them in `init` (as first actions):

```
msg0.ev= dial; msg0.param = 1;
msg1.ev= onhook; msg1.param = 0;
```

This allows us to define p and q as:

```
#define p uout[1]?[msg0]
#define q uout[1]?[msg1]
```

where p can be read as "the message on channel uout[1] is the structured message (*dial*, 0)" and similarly for q.

A run of the verifier (using supertrace) quickly returns with a sample trace where the claim is violated. Examining the trace, we can find the flaw: apart from receiving a busy-tone, the user can also go onhook – a behaviour which we wish to allow. So, the property that we should be considering is:

$$\Box(p \rightarrow ((s \wedge t)U(q \vee r) \tag{4}$$

with $r =$ *User 1 goes onhook* and the appropriate changes to the "constant" definitions and hash-define's.

Using supertrace mode we are able to verify the correctness of this property (equation 4) in less than 30 minutes. A first attempt to use exhaustive search [2] failed (out of memory) after nearly 10 hours, but did not find any violations. Reducing the estimated search depth reduced the memory required for supertrace by nearly 70 MB, though it did not have any impact on the time required. A second attempt of an exhaustive search, with a reduced stack, completed and confirmed the correctness of the property. Details can be obtained from Table 1. All runs look for assertion violations and invalid endstates, as well as the validity of the never claim. Partial Order Reduction was used throughout. The given times are CPU-times [3].

We note that an initialisation of the variables in the never-claim might be aesthetically more pleasing, but due to side-effect free behaviour of never-claims it is not allowed. A more elegant solution would be to simply define constants of structured types. This extension would be very helpful, as the property discussed here is rather simple in comparison with many others of interest. Such properties include many preconditions and complex relations between them, thus numerous global variables will be unavoidable.

Moreover, although the properties discussed here might not appear very interesting, once features, such as call forwarding, are added, their validity might change. Nevertheless, from a verification point they are already challenging – both in their formulation and in the use of resources during automatic verification.

[2] using -DMA420 and -DCOLLAPSE

[3] The results were obtained on a PentiumII-450 with 384MB physical memory and 384MB swap running Linux (Kernel 2.0.35)

Search Mode	Maximum Depth	Mem used (MB)	Actual Depth	States	Time (h:m:s)	coverage
supertrace	5000000	483392	2362771	3.8e+07	0:25:28	98 %
exhaustive	5000000	530981	1945846	3.3e+07	9:17:36	out of memory
supertrace	2400000	411514	2362771	3.8e+07	0:25:21	98 %
exhaustive	2400000	510463	2365719	3.8e+07	20:38:59	completed

Table 1: Results of Verification Attempts

7 Related and Further Work

Work on feature interaction is extensively described in the proceedings of the Feature Interaction Workshops (held regularly since the early 1990's and documented in [VGL92], [BV94], [CO95], [DBL97] and [KB98]) and in relevant publications of the telecommunications industry. So far no comprehensive methods have been developed that solve the problem. Much of the recent work concentrates on modelling features and their interaction detection to find automated solutions at design-time. At runtime, a reliance on having detected all errors at design-time is still common.

This paper has been based on a *network* view with the user as an observer; obviously *user* views are another way of modelling telephone switching systems. One advantage of a user view approach is further abstraction away from the underlying network; an example of this approach is given in [CM98]. Our ultimate goal is an online feature manager which can deal with proprietary (or legacy) systems, so we must work from the system perspective.

Based on the experience gained in this work we will include a feature manager and features (in the sense of [MM98]) in the model. Including the feature manager also implies that we develop algorithms for the detection and resolution of feature interactions; verification results of general and specific properties, will guide the algorithms.

Another line of further investigations is the idea of relativised temporal quantifiers (c.f. Section 6).

8 Conclusion

We conclude on three aspects of this work: the issues involved in modelling an operational system, the usage of Spin and Promela to assist with modelling and verification, and the overall aim of this work.

We have found that describing the network components as communicating finite state machines is natural and allows us to model on a rather abstract level, while at the same time it permits the possibility to extend the level of detail in any component.

Translating the resulting machines into PROMELA is relatively straightforward and the result remains faithful to the general structure of the model.

The experiments concerning the verification of specific properties showed that SPIN quickly shows when a property is not satisfied – this obviously does not give a correctness proof, but means that we can realistically use SPIN to find errors (and eliminate them).

Reasoning about structured channel values, as required for specific properties, leads us to believe that direct access to structured data values would be beneficial. Artificial constants and extra variables (which extend the state space) would then be superfluous.

Our verification results indicate that SPIN as a verification and simulation tool should prove helpful for our overall research aims, namely the verification of the model extended by a feature manager and features.

References

[BV94] L. G. Bouma and H. Velthuijsen, editors. *Proceedings of the* 2^{nd} *International Workshop on Feature Interactions in Telecommunications Systems.* IOS Press (Amsterdam), May 1994.

[CM98] M. Calder and A. Miller. Analysing a basic call protocol using PROMELA/XSPIN. In *Proceedings* SPIN*'98: Workshop on automata theoretic verification with the* SPIN *model checker*, pages 169–181, Paris, France, November 1998.

[CMM99] M. Calder, E. Magill, and D. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *To appear in IEE Proceedings - Software*, 1999.

[CO95] K. E. Cheng and T. Ohta, editors. *Proceedings of the* 3^{rd} *International Workshop on Feature Interactions in Telecommunications Systems.* IOS Press (Amsterdam), October 1995.

[DBL97] P. Dini, R. Boutaba, and L. Logrippo, editors. *Proceedings of the* 4^{th} *International Workshop on Feature Interactions in Telecommunications Systems.* IOS Press (Amsterdam), June 1997.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall (New York), 1991.

[Hol93] G. J. Holzmann. Desgn and validation of protocols: A tutorial. *Computer Networks and ISDN Systems*, pages 981–1017, 1993.

[KB98] K. Kimbler and L. G. Bouma, editors. *Proceedings of the* 5^{th} *International Workshop on Feature Interactions in Telecommunications and Software Systems.* IOS Press (Amsterdam), September 1998.

[LT88] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. November 1988.

[MM98] D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In K. Kimbler and L. G. Bouma, editors, *Proceedings of the* 5^{th} *International Workshop on Feature Interactions in Telecommunications and Software Systems*, pages 115–134, Lund, Sweden, September 1998.

[RV94] F. J. Redmill and A. R. Valdar. *SPC: Digital Telephone Exchanges*. Peter Peregrinus Ltd. (Stevenage), 1994.

[VGL92] H. Velthuijsen, N. Griffith, and Y.-J. Lin, editors. *Proceedings of the* 1^{st} *International Workshop on Feature Interactions in Telecommunications Software Systems*, December 1992.