Analysing a Basic Call Protocol using Promela/XSpin

Muffy Calder and Alice Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.

Abstract

A basic call model for telecommunications services, including a communication protocol for asynchronous communication between call processes, is defined in Promela. The model and protocol are analysed for using XSpinand some errors are uncovered.

keywords

telecommunications services and features; Promela/XSpin; communicating processes; formal modelling; analysis and reasoning techniques.

1 Introduction

As telecommunications services become implemented in software, and consequently increasingly complex, it is important to be able to analyse aspects of their behaviour. For example, it may be important to detect and/or resolve any interactions between the features of one or more services [1]. Given the complexity of the domain, and the large number of features and services, automated analysis is not only desirable, but necessary. A number of formalisms and tools have been applied, including SDL and Lotos, e.g. see a number of papers in [2, 3, 4]. However, many attempts at analysis suffer from a state-space explosion, and therefore the authors end up compromising either in the level of abstraction of the model, the number of processes in a network, or complete exploration of the state space.

This paper reports on a preliminary investigation of the feasability of using Promela/XSpin [11] for modelling and analysing telecommunications services. Our motivation derives from our experiences with other formalisms and modelling approaches (e.g. [13]), and the pitfalls therein.

Specifically, our aim is to develop the basic call model of POTS (*Plain Old Telephony System*) in Promela, and to perform analysis on a network of call processes. While this goal is not novel, we have chosen a level of abstraction which is close to an operational one and has asynchronous inter-process communication. These are two well-known sources of difficulties for analysis. Given this context, we therefore tried to introduce as little atomicity and/or synchronicity as possible into the Promela implementation.

The paper is organised as follows. The Basic call model is informally described in the next section. In section 3 the communication protocol is described and section 4 contains the Promela implementation. The following section, 5, summarises the analysis; related work is discussed in 6 and our conclusions follow in 7.

2 Overview of the Basic Call Model

While there is no one standard definition of the basic call process of POTS, we loosely follow the IN [8] (*Intelligent Networks*) approach.

At our level of abstraction, each user is identified with a call process and we make no distinction between numbers and terminal devices. We also consider numbers as single entities: we are not concerned with individual digits.

In each call process we distinguish between *originating party* behaviour (the party which initiates the call) and the *terminating party* behaviour (the party which is called). Thus the call process state space can be divided into two subspaces: originating party states and terminating party states. An overview of the call model is given by a state transition diagram in figure 1. The originating party states are in the lower half of the diagram, and include the states diall, calling, unobt, olaert, busy, oconnected, oringout, otalk, and oclose. The terminating party states are in the upper half of the diagram, and include the states talert, tpickup, tconnected, and ttalk. The state idle belongs to both originating and terminating behaviour.

The transitions are labelled by user events, i.e. those events which are initiated by a user, such as *off* (handset off), *on* (handset on), etc. or are perceived by the user, such as *oring* (the originating party hears ringing tone on line) and *tring* (the terminating party's device rings).

Note that there is an assymmetry between originating and terminating behaviour, with respect to call clear down: only the originating party can initiate close down. Thus it is important to distinguish between originating and terminating connected and talk states.

Call behaviour is determined both by a user's actions *and* by other users' actions and the network. Therefore, some form of communication between processes, and a protocol determining call set up and clear down, is required. These are described below; note that this aspect is *not* reflected in figure 1.

3 The Protocol

Each user call process has an associated channel, called a communication channel, with capacity for one message (or *token*). The tokens will in fact be the channels themselves. For user process A, we will refer to its associated communication channel as channel A.

When a communication channel is empty, then that user is not connected to, or attempting to connect to, any other user. We will refer to this situation as the user is not engaged in a call; when a communication channel is not empty, then we say that the user is engaged in a call (but not necessarily connected to another user). When channel A contains channel B, and channel B contains channel A, then a connection has been established between users A and B.

The basic protocol for call setup is as follows. Assume that user A wants to connect to user B, (i.e. A rings B) and A and B are both not engaged in a call. When A goes off hook, A immediately places a token (itself) on its communication channel. After dialling B, A sends a token (itself) to channel B. When B receives the token on its communication channel, it sends a token (itself) to channel A. When channels A and B contain each other, the connection is established.

The basic protocol for call clear down is as follows. Assume that users A and B are connected and that A is the originating party. A can close down one side of the connection by going on hook and removing the token from channel A. Channel B remains the same, since B is still engaged in a call, though not connected. (i.e. containing a token) When channel A is empty, then B can also go on hook and

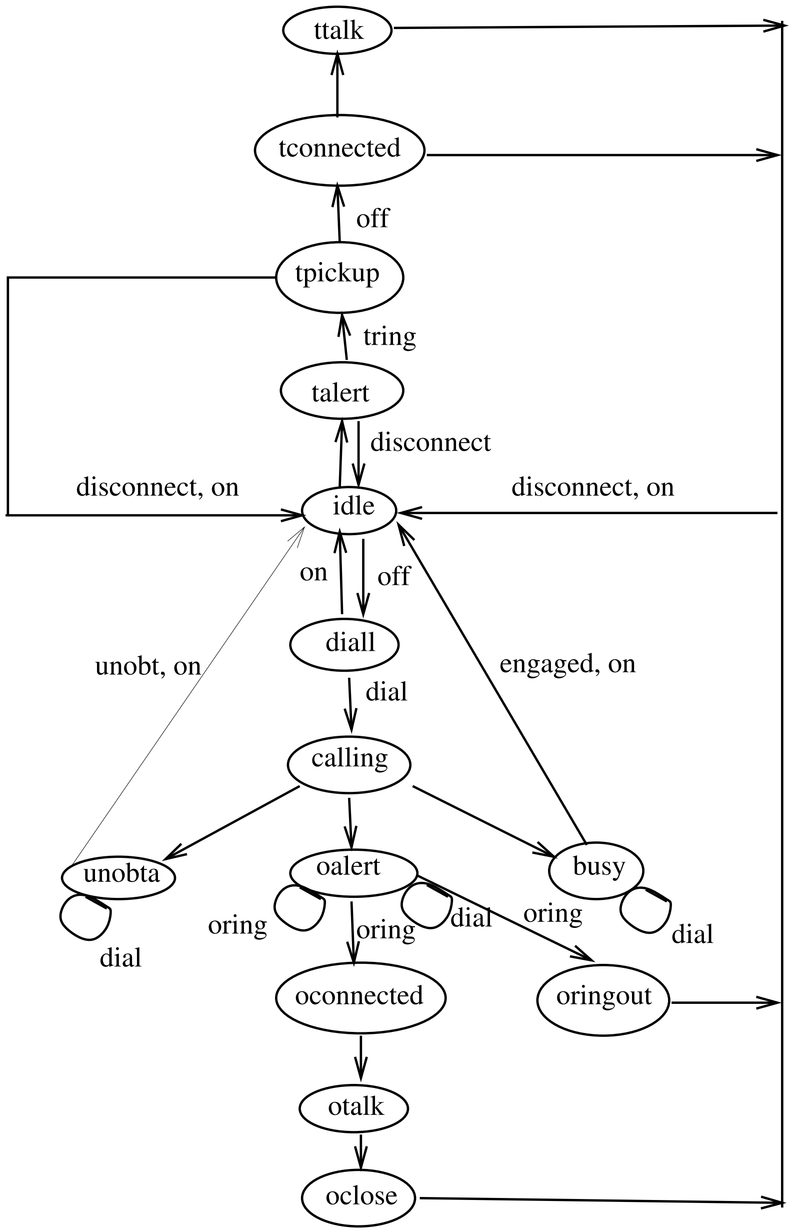

Figure 1: Basic Call - States and Events

remove the token from its channel. (If B goes on hook first, i.e. before channel A is cleared, then the action will have no effect.)

3.1 Additional Synchronisation Variables

While the basic protocol and state transitions are reasonably straightforward, the main difficulty in developing this model is with call cancellation, or "non-standard" behaviour during call setup, and the race conditions which can result. It is important to note that these race conditions were uncovered only through extensive simulation and (attempted) verification, and XSPINwas invaluable in uncovering them. To overcome them, we added a synchronisation variable for each user process. While other solutions might be possible, we found this to be a simple approach, and one which does not further over-constrain the concurrency.

4 The Basic Call Model in Promela

A user call process is given by the (parameterised) process "user"; the network is given by four instantiations of that process.

Global types and variables are defined below, the following section contains the process "User".

4.1 Types and variables

The four user communication channels are given by the channels "one" to "four" and the associated synchronisation variables are given by "sync1'" to "sync4", respectively. The variables "event1" to "event4" and "partner1" to "partner4" are global variables used only for the purposes of verification (see the following section for their use). The "connect" variables are predicates used to denote the presence or absence of a connection: "connectA[B]" denotes a connection from user A to user B (i.e. A is the originating party).

```
mtype = {on,off,dial,disconnect,oring,tring,unobt,engaged,
         speech,hung_up,connected};

chan one   = [1] of {chan};
chan two   = [1] of {chan};
chan three = [1] of {chan};
chan four  = [1] of {chan};

chan sync1 = [0] of {mtype};
chan sync2 = [0] of {mtype};
chan sync3 = [0] of {mtype};
chan sync4 = [0] of {mtype};
chan null  = [1] of {chan};

mtype event1 = on;
mtype event2 = on;
mtype event3 = on;
mtype event4 = on;

chan partner1 = null;
chan partner2 = null;
chan partner3 = null;
chan partner4 = null;
```

```
byte connect1[5];
byte connect2[5];
byte connect3[5];
byte connect4[5];

chan y = [1] of {chan};
```

4.2 The User Process

The user call process has 10 parameters: self, linea, lineb, linec, sync, synca, syncb, syncc, partner, and event. The first four are the user communication channels, "self" being the user and "linea" ... "linec" being the other users. There are four synchronisation variables, with "sync" belonging to "self". "partner" is a (channel) variable which denotes the current partner (set to "null" if the user is not engaged in a call) and "event" is a variable which denotes the most recent event. Local variables include "dev" for the state of the device (e.g. on, off) and "z" and "x" for communication purposes.

Note that since in PROMELA reading from a channel is destructive, a non-destructive read is implemented by an atomic read following by a write. Also, a drawback of our simple treatment of channel variables as disjoint variables is that sometimes the value of a (channel) variable determines another channel to be selected. For example, if "x == linea" then the process should should synchronise on channel "synca". Unfortunately, in the current implementation we have no facility for allowing channel names to index into a collection of channels; in a future implementation we will try to employ arrays to avoid this problem.

It is important to note that here are numerous assertions within the code, particularly at points when entering a new (call) state, and when reading and writing to communication channels. These assertions proved invaluable for debugging and form an essential part of the verifiction process.

In the body of the process, comments, in general, follow the code.

```
proctype User (chan self,linea,lineb,linec,sync,synca,syncb,syncc,
                partner; mtype event)
{

mtype dev = on;
chan x = [1] of {chan};
mtype z= on;


idle:assert(dev == on && partner==null);
     if
     :: atomic{empty(self)-> event = off; dev = off; self!self}; goto diall
        /* no connection is being attempted, go offhook */
        /* and become originating party */

     :: atomic{full(self)->self?partner; self!partner};
       /* a connection is being attempted */
       /* either become terminating party */
       /* or connection attempt is cancelled and remain in idle state */
        if
        :: partner==linea;
```

```
            if
            :: atomic{synca?y; self?x; partner=null; goto idle}
            :: goto talert
            fi
         :: partner==lineb;
            if
            :: atomic{syncb?y; self?x; partner=null; goto idle}
            :: goto talert
            fi
         :: partner==linec;
            if
            :: atomic{syncc?y; self?x; partner=null; goto idle}
            :: goto talert
            fi
        fi
     fi;


diall: assert(dev == off && full(self)&& partner==null);
       if
       :: event = dial;
          /* dial and then nondeterministic choice of called party */
         if
         :: partner = self; goto calling
         :: partner = linea; goto calling
         :: partner = lineb; goto calling
         :: partner = linec; goto calling
       fi
       :: atomic{event = on; dev = on; self?x; goto idle}
          /*go onhook, without dialling */
       fi;


calling: assert(dev == off);
         if
         :: partner == null -> goto unobta
            /* invalid partner */
         :: partner == self -> goto busy
            /* invalid partner */
         :: else -> if
                    :: partner!self -> goto oalert
                      /* valid partner, write token to partner's channel*/
                    :: full(partner) -> goto busy
                  /* valid partner but engaged */
                   fi
         fi;


unobta: assert(full(self));
        event = unobt;
        if
        :: event = dial; goto unobta
           /* trivial dial */
        :: atomic{event = on; dev = on; partner = null;
```

```
                  self?x; assert (x==self); goto idle}
           /* go onhook, cancel connection attempt */
        fi;


busy: assert(full(self));
      event = engaged;
      if
      :: event = dial; goto busy
         /* trivial dial */
      :: atomic{event = on; dev = on; partner = null;
                self?x; assert (x==self);goto idle}
         /*go onhook, cancel connection attempt */
      fi;


oalert: assert(full(partner) && full(self) && dev == off);
        event = oring; atomic{self?x;self!x}
        /* check channel */
        if
        :: (x == partner) -> goto oconnected
           /* correct token */
        :: (x != partner) -> goto oalert
           /* wrong token, try again */
        :: (x != partner) -> goto oringout
           /* wrong token, give up */
        :: event = dial; goto oalert
           /* trivial dial */
        fi;


oringout:
        atomic{event = on; dev = on;partner = null};sync!hung_up;
        atomic{self?x;goto idle};
        /* give up, go onhook */



oconnected: assert (full(self) && full(partner));
            atomic{sync!connected; printf("oconnected "); goto otalk};


otalk: assert (full(self) && full(partner));
       /* connection established */
       if
          ::  ((self==one)&&(partner==two))->connect1[2]=1;
          ::  ((self==one)&&(partner==three))->connect1[3]=1;
          ::  ((self==one)&&(partner==four))->connect1[4]=1;

          ::  ((self==two)&&(partner==one))->connect2[1]=1;
          ::  ((self==two)&&(partner==three))->connect2[3]=1;
          ::  ((self==two)&&(partner==four))->connect2[4]=1;

          ::  ((self==three)&&(partner==one))->connect3[1]=1;
```

```
          :: ((self==three)&&(partner==two))->connect3[2]=1;
          :: ((self==three)&&(partner==four))->connect3[4]=1;

          :: ((self==four)&&(partner==one))->connect4[1]=1;
          :: ((self==four)&&(partner==two))->connect4[2]=1;
          :: ((self==four)&&(partner==three))->connect4[3]=1;
       fi;
       goto oclose;


oclose: assert (full(self) && full(partner));
        /* disconnect */
        if
          :: ((self==one)&&(partner==two))->connect1[2]=0;
          :: ((self==one)&&(partner==three))->connect1[3]=0;
          :: ((self==one)&&(partner==four))->connect1[4]=0;

          :: ((self==two)&&(partner==one))->connect2[1]=0;
          :: ((self==two)&&(partner==three))->connect2[3]=0;
          :: ((self==two)&&(partner==four))->connect2[4]=0;

          :: ((self==three)&&(partner==one))->connect3[1]=0;
          :: ((self==three)&&(partner==two))->connect3[2]=0;
          :: ((self==three)&&(partner==four))->connect3[4]=0;

          :: ((self==four)&&(partner==one))->connect4[1]=0;
          :: ((self==four)&&(partner==two))->connect4[2]=0;
          :: ((self==four)&&(partner==three))->connect4[3]=0;
        fi;

        atomic{event = on; dev = on;
               self?x; assert (x == partner);
               partner?x; partner!x; assert ( x == self);
               partner = null;
               goto idle};
              /* empty own channel and check partner's channel*/


talert: assert(dev == on);
        /* either connection attempt is cancelled and then empty channel */
        /* or device rings */
        if
        :: partner==linea ->
           if
           :: atomic{synca?z; event = disconnect};
              self?x; partner=null; goto idle
           :: event=tring; goto tpickup
           fi
         :: partner==lineb ->
            if
            :: atomic{syncb?z; event = disconnect};
               self?x; partner=null; goto idle
            :: event=tring; goto tpickup
            fi
```

```
         :: partner==linec ->
            if
            :: atomic{syncc?z; event = disconnect};
               self?x; partner=null; goto idle
            :: event=tring; goto tpickup
           fi
         fi;


tpickup: atomic{event = off; dev = off};
         /* either connection attempt is cancelled and then empty channel */
         /* or proceed with connection and write token to partner's channel */
        if
        :: partner==linea->
           if
           ::atomic{synca?z;event = disconnect; event = on; dev = on};
             atomic{self?x;partner=null;goto idle}
           ::atomic{partner?x;partner!self;goto tconnected}
           fi
        :: partner==lineb->
           if
           :: atomic{syncb?z;event = disconnect; event = on; dev = on};
              atomic{self?x;partner=null;goto idle}
           :: atomic{partner?x;partner!self;goto tconnected}
           fi
        :: partner==linec->
           if
           :: atomic{syncc?z;event = disconnect; event = on; dev = on};
             atomic{self?x;partner=null;goto idle}
           :: atomic{partner?x;partner!self;goto tconnected}
           fi
        fi;


tconnected:  /*read sync variable */
             if
             :: partner==linea->synca?z
             :: partner==lineb->syncb?z
             :: partner==linec->syncc?z
             fi;
             if
             :: atomic{z==connected->printf("tconnected ");goto ttalk}
                /* connection established */
             :: atomic{z==hung_up->event = on; dev = on};
                atomic{self?x;assert(x==partner);partner=null;goto idle}
                /* connection attempt is cancelled, empty channel */
     fi;


ttalk: if
       :: atomic{empty(partner) -> event=on; dev=on;
          self?x; assert(x==partner); partner = null; goto idle}
          /* connection is cancelled by originating party */
          /* go onhook and empty channel */
```

```
    fi;


} /* end User */
```

4.3 Networks of Users

Any number of call processes can be run concurrently. For example, a "network" of four user call processes is given by the following instantiations of the process "User".

```
init
{
      atomic{
      p1=run User(one,two,three,four,sync1,sync2,sync3,sync4,partner1,event1);
      p2=run  User(two,three,four,one, sync2,sync3,sync4,sync1,partner2,event2);
      p3=run  User(three,four,one,two,sync3,sync4,sync1,sync2,partner3,event3);
      p4=run  User(four,one,two,three,sync4,sync1,sync2,sync3,partner4,event4);
     }
}
```

5 Analysis and Discussion

The asynchronous nature of the communication between call processes and the intricacies of call setup and cleardown give rise to plenty of scope for errors in the underlying protocol. In particular, it can be difficult to handle call cancellation at an advanced stage correctly, and it is quite easy to make incorrect assumptions about the state of a communication channel or to overlook an unexpected race condition.

Therefore, an important part of the verification process is to perform an "behavioural audit" as the call progresses. To this end, assertions are embedded in the code at key points such as at the beginning of a new call state and when reading (destructively, or otherwise) from a communication channel.

The former take the form of predicates about the state of the device, the last event performed, and the presence or absence of a token on the communication channels of the parties involved in the connection (attempt). For example, the "oalert" state begins with "oalert: assert(full(partner) && full(self) && dev == off)".

The latter take the form of comparing a newly read value of a channel variable with the expected value; for example "self?x; assert(x==partner)".

5.1 Further Properties

Further temporal properties are defined using LTL and satisfied (or otherwise) using the *never* claims facility.

There are numerous properties that we could demonstrate. We will not enumerate them all here, but only discuss two of the more interesting ones. Informally, they are: *a connection between two users is always possible*, and *if you dial yourself, then you will hear the engaged tone.*

In our implementation, these properties have to be expressed with respect to particular users. So, for example, instantiations of the first property are:

$\langle\rangle$connect1[2]== 1,
$\langle\rangle$connect2[1]== 0, ...
Instantiations of the second property are
[](partner1 == one $\rightarrow$ $\langle\rangle$event1 == engaged),
[](partner2 == one $\rightarrow$ $\langle\rangle$event2 == engaged), ...

5.2 The Two User Model

In the interests of simplicity, we began our debugging and analysis with a cut-down model involving only two user processes. (At this stage, there were no synchronisation variables present.) Assertion checking proved invaluable, as we uncovered several genuine bugs in the protocol, as well as numerous small, but non-trivial clerical errors.

Also, we found that we had to introduce a few additional atomic clauses, to prevent race conditions.

Once the assertions were checked (exhaustively), the temporal properties were easily satisfied.

5.3 The Four User Model

We then moved on to a four user process model. This proved to be quite interesting because during an attempt to violate a never claim (for one of the temporal formulae) we found that an assertion was violated! After some examination of the traces, it was discovered that we had overlooked a race condition during call cancellation.

It is important to note that the erroroneous scenario can *not* be created with only two call processes; at least three call processes are required to create the (potentially erroneous) circumstances.

The error was as follows. Suppose user 2 is calling user 1, the latter is idle. Some time later, user 1 is in state "talert", user 2 is in state "oringout" and user 2 removes the tokens from both channels 1 and 2 and proceeds to idle. (In the original version, tokens were removed in "oringout" from both originating and terminating party's communications channels.) But this leaves the possibility that another user process, e.g. user 3, can "sneak" in and leave a token on channel 1, and so because user 1's channel is full, it proceeds as if it is still trying to set up a connection with user 2. Eventually, user 1 writes a token to user 2's channel, when user 2 is not expecting another token.

As a result of this situation, we introduced the synchronisation variables.

The full four user model could not be checked exhaustively on a 64 MB machine. Unfortunately, we also had trouble using the supertrace option, and could only demonstrate that a connection was possible between users 1 and 2 (the other permutations led to memory exhaustion). Presumably, a reordering of the initialisations would enable the other permutations to succeed.

A small improvement (suggested by Rob Gerth) was gained by expressing eventually formulae of the form $\langle\rangle f$ as $!fUf$.

5.4 The Three User Model

Finally, we moved to a three user model to check the safety property

[](partner1 == one $\rightarrow$ $\langle\rangle$event1 == engaged). This succeeded at a depth of over 16K, with state vector 155 bytes and 260K states.

6 Related Work

While PROMELA and XSPIN have already been demonstrated to be useful in the field of modelling and analysing feature "building blocks" [10], our motivation here is rather more similar to the approaches taken in [12] and by the first author (of this paper) in [13].

Both of these papers extend the basic call functionality to include several features. However, both have synchronous communication between user processes, thus cutting down on the state space (and complexity of protocol) significantly. The call model in [12] is also rather more abstract, and therefore does not involve the numerous possibilites for incorrect call cancellation. They use the SMV language/checker, but eventually suffer from the state-explosion problem and consider only 2 complete processes, plus one originating only and one terminating only. On the other hand, they have expressed and tested numerous interesting properties, and we would like to be able to conduct similar experiments. Our earlier work ([13]), using LOTOS and the model-checker of [7], employs a similar level of abstraction and includes seven further features, but it too has synchronous communication. We were able to perform some analysis of μ-calculus formulae, with fixed-points, but they were only very simple properties, and took 19 hours to satisfy! Satisfaction of all other properties proved to be intractable and subsequently we decided to explore the use of other formalisms and tools.

7 Conclusions

We have defined a basic call model and an underlying communication protocol in PROMELA. The model is quite close to the operational level and so it was non-trivial to get the protocol correct. While we were a little disappointed that we were unable to check all properties on the four user model, using XSPIN we were able to uncover some important errors. It was crucial to be able to consider more than two call processes to uncover them, and clearly the capabilities of XSPIN were required for this.

Our confidence in the model is vastly improved as a consequence, and we can consider adding further features. It remains to investigate techniques for dealing with the further properties for networks of more than three call processes (and also to determine exactly when more than three are necessary). In particular, now that we are confident that our protocol is correct, we will obtain the necessary resources to carry out further experimentation to check all the temporal properties.

Acknowledgements

We would like to thank Rob Gerth for his help and advice on using XSPIN.

References

[1] A. Alfred, N. Griffith. Feature Interactions in the Global Information Infrastructure. In Proceedings of 3rd ACM Sigsoft Symp. on Foundations of Software Engineering, *Software Engineering Notes*, Vol. 20, No. 4, Oct. 1995.

[2] Proceedings of International Workshop on Feature Interactions in Telecommunications Systems II, St. Petersburg, U.S.A., IEEE Communications Society, 1992.

[3] W.Bouma and H.Velthuijsen (eds.). *Feature Interactions in Telecommunications Systems II.* Proceedings of International Workshop, Amsterdam, IOS Press, 1994.

[4] K.E. Cheng and T. Ohta (eds.). *Feature Interactions in Telecommunications Systems III.* Tokyo, IOS Press, 1995.

[5] P. Dini, R. Boutaba, and L. Logrippo, (eds.) *Feature Interactions in Telecommunications Systems IV.* Montreal, IOS Press, 1997.

[6] E.J. Cameron, N.D. Griffeth, Y.J. Lin, M.E. Nilson, W.K. Shnure, and H. Velthuijsen. A feature interaction benchmark in IN and beyond. In [3].

[7] H. Garavel et al. CAESAR toolkit. Available from Hubert.Garavel@imag.fr.

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

[9] *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International Organisation for Standardisation. 1988.

[10] F.J.Lin and Y-J. Lin. *A Building Block Approach to Detecting and Resolving Feature Interactions*, in [3].

[11] G. Holzmann. Design and Validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, No. 25, pp. 981-1017, 1993.

[12] M. Plath and M. Ryan. Plug-and-play features. To appear in *Feature Interactions in Telecommunications Systems V.* IOS Press, 1998.

[13] M. Thomas. Modelling and Analysing User Views of Telecommunications Services In [5].