Feature validation for any number of processes

M. Calder and A. Miller

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

May 2002

Glasgow University Department of Computing Science
Technical Report No.TR-2002-10

Abstract

The automatic verification of concurrent systems by model-checking has traditionally been limited due to the inability to generalise results to systems consisting of any number of processes. In this paper we show how a technique based on abstraction (rather than induction) can be used to prove general results about a concurrent system of, say, \( n \) processes. The key idea is to consider a system of constant number \( (m) \) of concurrent processes, in parallel with one “abstract” process which represents the product of any number of other processes.

The application domain is feature interaction analysis of telecommunications service with features. The initial model on \( m \) processes (for any specified set of selected features) is automatically generated from a template, using Perl scripts. Similarly, the conversion from an \( m+1 \) process model to an \( m \) process plus abstract model, representing an arbitrary \( n \) process model, is performed automatically.

1 Introduction

The automatic verification of concurrent systems by model-checking has traditionally been limited due to the inability to generalise results to systems consisting of any number of processes. For example, we may be able to show that a property holds for 4 or 5 concurrent processes, i.e. for \( p_0 \parallel p_1 \parallel p_2 \parallel p_3 \) or for \( p_0 \parallel p_1 \parallel p_2 \parallel p_3 \parallel p_4 \), but how can we deduce that (if at all) the property holds
$p_0||p_1||p_2||p_3|\ldots||p_{n-1}$, for an arbitrary $n$? It is not possible to demonstrate this with straight-forward model-checking [1].

Recently, attempts have been made to overcome this problem using methods based on a combination of abstraction and model-checking [13, 18] or of induction and model-checking [8, 15]. In this paper we show how the first of these approaches can be used to prove general results about a concurrent system.

Our application domain is feature interaction analysis [2]. The analysis involves examining a system of basic service processes, running concurrently, some with additional features, to determine whether or not certain properties hold. Briefly, an interaction may be indicated when a property holds in the presence of one feature alone (i.e. all but one process offers only the basic service) but is violated in the presence of more than one feature. It is important to know when analysis results do (and do not) scale up.

For example, for a system of four concurrent service processes, with different combinations of pairs of features, the goal of the analysis is to prove (or disprove) that $M(p_0||p_1||p_2||p_3) \models \phi[0,1,\ldots,t]$ where $M(p_0||p_1||p_2||p_3)$ is the finite-state model of the parallel composition of processes $p_0, p_1, p_2, p_3$ (instances of a parameterised process $p$) and $\phi[0,1,\ldots,t]$ is a temporal logic formula containing free variables indexed by $0, 1, \ldots, t$, where $0 \leq t \leq 2$. The $p_i$ communicate peer to peer, asynchronously. In general, the $p_i$ are not isomorphic (because they have different sets of features enabled).

We have demonstrated a number of such results [7, 4, 5] for a basic telecommunications service with features modelled in Promela. The properties are specified in linear temporal logic (LTL) and verified using the SPIN model checker [11]. The $\phi[0,1,\ldots,t]$ express properties about feature behaviour (e.g. if process 1 has call forwarding to process 2 and process 0 initiates a call to process 1, then eventually a call from process 0 to process 2 will be attempted). In some cases it is necessary to consider more than 4 processes (up to 6) to fully capture all possible combinations.

The problem is how to generalise such results to any number of concurrent, communicating processes, i.e. to demonstrate that the property $\phi$ holds regardless of the number of processes involved (providing this number is sufficiently large). In this paper, we offer a solution based on abstraction and model-checking.

Specifically, we give a technique to prove that, for a fixed $m$ and $0 \leq t \leq m-1$, for any $n$, if $p_m, p_{m+1}, \ldots, p_{n-1}$ are isomorphic (they have no features
enabled), then

\[ M(p_0 || p_1 || p_2 || p_3 \ldots || p_{n-1}) \models \phi[0, 1, \ldots, t]. \]

The technique involves representing the behaviour of \( p_m || \ldots || p_{n-1} \) by an abstract process, \( Abs \). A model of the \( m \) concrete processes \( p_0, p_1, \ldots, p_{m-1} \) together with the abstract process is generated automatically from a model of the concrete processes together with a single basic call process \( p \). For example if \( m = 3 \), a model of \( p_0 || \ldots || p_2 || Abs \) is generated from a model of \( p_0 || \ldots || p_2 || p \). In this case, for any \( 0 \leq t \leq 2 \) we show, by model-checking, that \( M(p_0 || p_1 || p_2 || Abs) \models \phi[0, \ldots, t] \) and can hence infer that

\( \forall n. M(p_0 || p_1 || p_2 || p_3 \ldots || p_{n-1}) \models \phi[0, \ldots, t]. \)

The technique is summarised by Figure 1.

The concrete processes are \( p_0, p_1, p_2 \). Communication between concrete processes is via channels (denoted by rectangles) and is unchanged. Note that the contents of a communications channel fundamentally determines process behaviour. The observable behaviour of processes \( p_3 \ldots p_{n-1} \) is represented by the abstract process in the following way. Communication to/from a concrete process from/to any other process takes place via a virtual channel. Rather than concrete processes reading/writing to this (virtual) channel and behaving accordingly, each possible read is replaced by a non-deterministic choice over the possible contents of such a channel. In this way all possible behaviours are explored. (A write to such a channel is no longer relevant.)

The number of concrete processes that are required depends on the number of distinct variables that occur in the feature descriptions and the property to be verified. In section 6.3, we show how, if two features are enabled
in total, then for our specific set of properties, at most 5 concrete processes are required (that is, \( m \leq 5 \)).

The generation of the initial model on \( m + 1 \) processes (for any specified set of selected features) is automatically generated from a template, using Perl scripts. Similarly, the conversion from an \( m + 1 \) process model to an \( m \) process plus abstract model is performed automatically.

The purpose of this report is to provide a more detailed description of the approach described in recently submitted work [5, 6], via the inclusion of greater implementation detail, examples and scripts.

2 The basic call service and features

2.1 The basic call

The basic call service permits call set-up and tear-down between two parties. Call control is asymmetric: one party has originating behaviour, and controls the call, the other has terminating behaviour. Our model is that described in previous work [4, 5] and follows the IN (Intelligent Networks) model, distributed functional plane [12].

Figure 2 gives a diagrammatic representation of an abstract automaton for the basic call service behaviour (note the full implementation is somewhat more complicated). States to the left of the idle state represent terminating behaviour, states to the right represent originating behaviour. Transitions between states are triggered by user-initiated events at the terminal device, such as (handset) on and (handset) off, or by communication events on shared channels. We have excluded some trivial behaviour from the automaton. For example, it is possible to perform a dial event (with no effect) from most states. Note also that while the state preidle is an important detail of the implementation (where local and global variables are reset), it does not play a part in the observable behaviour of a call process.

Originating and terminating automata influence each other’s behaviour through communication via (shared) channels. In the automaton, the channels are referred to as \( c \), for the channel associated with that process, and \( p \), for the channel associated with the partner process. \( p \) is chosen non-deterministically. We use the notation \( c!x, y \) to denote write the value \((x, y)\) to the channel \( c \), \( c!!x, y \) to denote overwrite the channel \( c \) with \((x, y)\), \( c? < x, y > \) to denote poll or non-destructively read value \((x, y)\) from channel \( c \),
and $c?x, y$ to denote destructively read value $(x, y)$ from channel $c$. When the value may be arbitrary, we use variables $x$ and $y$; otherwise we use the actual constants required, e.g. 0, 1, $p$, etc. Note that channels may contain channels.

Each channel has capacity for at most one message: a pair consisting of a channel name (the other party in the call) and a status bit (the status of the connection). 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 3.

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, (and after polling to ensure that A has not rung off) the message $(B, 1)$ is sent to channel A and the status bit in the message on channel B is changed to 1; the connection is then established. To clear down, A can close down one side of the connection by going on hook: the message is removed from its communication channel and the status bit of the message in channel B becomes 0. Then, neither A nor B are in a connected state, and A is free to close down the connection. On the other hand, channel B cannot close down the connection (reflecting the real-life situation). So, if B goes on hook, while A and B are connected, then the connection status remains unchanged for both A and B.

The automaton is not a complete specification, but gives an indication of observable behaviour and the role of communication between processes.
<table>
<thead>
<tr>
<th>Contents of Channel A</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>A is free</td>
</tr>
<tr>
<td>(A,0)</td>
<td>A is engaged, but not connected</td>
</tr>
</tbody>
</table>
| (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 |

Figure 3: States of a communications channel in the protocol

2.2 Features

We consider a set of 9 features, each of which have a corresponding LTL property associated with them. As the purpose of this paper is not to describe the features but to use them to illustrate our abstraction process, we describe the features in English below, but do not provide detailed descriptions or LTL properties for all of them. Details of the first 5 of these features are given in earlier work [3] and [4], (though the latter considered fewer features and a different implementation of RBWF). However, details of these features are not important here.

- **CFU** call forward unconditional
- **CFB** call forward when busy
- **OCS** originating call screening
- **ODS** originating dial screening
- **TCS** terminating call screening
- **RBWF** ring back when free
- **OCO** originating calls only
- **TCO** terminating calls only
- **RWF** return when free
We will refer to the CFU feature throughout this paper. For illustration, we provide the LTL formula, but not the detailed description of the process User, its variables and associated propositions. Note that att(i, k) denotes the proposition “a call is attempted from User[i] to User[j]” which is itself defined in terms of the global variables associated with the User process. Also, User[i] refers to the User process with id i (see below).

Property 1 – CFU Assume that User[j] forwards to User[k]. If User[i] rings User[j] then a connection between i and k will be attempted before User[i] hangs up.
LTL: \[ [(p \rightarrow (rUq))] \]
\[ p = ((\text{dialed}[i] == j) \&\& (\text{User}[i]@\text{calling})), \]
\[ r = \text{att}(i, k), q = (\text{dev}[i] == \text{on}). \]

2.3 The basic call with features in Promela

For a full description of Spin and Promela refer to [11]. Here we give a brief description of pertinent aspects of a model consisting of 4 basic call processes. Due to space restrictions we give virtually no details of the Promela code associated with the basic call. Instead we refer the reader to previous work [4, 5].

The basic model for 4 user processes consists of four instantiations of the parameterised proctype User. The parameters associated with such a process are self (the channel associated with that process) and selfid (the id of that process). The code for each User proctype is separated into the call states given in figure 2, with the code associated with each all state grouped together as an atomic statement, concluding with the appropriate goto (to the next call state). Local and global variables are updated at various points in the call. Features are implemented within the Promela code via an inline function (a procedure with dynamic bindings used in SPIN) – feature\_lookup.

The relevant section of the feature\_lookup function when only a CFU feature is present is given below:

```promela
inline feature\_lookup (q1, id1, st)
{
  do
  ::((id1!=7)&&(st==st\_dia1))\&\&(CPU[id1]!=6))
  ->id1=CPU[1d1];
  q1=chan\_name[id1]
  ::else->break
  od
}
```

7
3 Feature validation and interaction analysis for a constant number of processes

The basic idea of feature interaction analysis is detecting when features behave as expected in isolation, but not in the presence of each other. So, interaction analysis involves feature validation (checking a feature in isolation) and then analysis of tuples of features (checking for violation of expected behaviour). This sounds like a combinatorial nightmare: we have a number of features, and exponential number of feature tuples, and a countably infinite number of user processes. Fortunately, we need only restrict our attention to pairwise analysis, as empirical evidence shows that it is extremely rare to have a 3-way interaction which is not detected as 2-way interaction [14].

In previous work [5] all of the features described above have been validated, for systems of four processes. That is, it has been proved, using model checking, that for each feature, for all models representing four call processes in which one user has that feature enabled (and no other process have any features enabled), the associated property holds. For each combination of feature and associated property, a relevant model needs to be individually constructed, to ensure that only relevant variables are included and set (to limit the size of the state-space). These models are generated using a Perl script.

Similarly, pairwise feature analysis has been performed in the four process case [5].

In the following section we discuss how these results can be extended to systems of $n$ processes.

4 Verification of any number of call processes

For a given number of features present, and the property to be verified, there is a fixed number of processes $m$ for which proof of the property is sufficient to prove the property for any number of processes. For example, suppose we wish to prove the (CFU) property

If User [0] rings User [1] then a connection between User [0] and User [2] will be attempted before User [0] hangs up

for the model representing the behaviour of User [0]|User [1]|...|User [$n-1$] such that User [1] has call forward unconditional to User [2] and the other
users are basic call processes, for any \( n \). Proof of the property for \( n = 4 \) seems to be sufficient (the three processes involved in the feature and the property, plus an external process). But why and when is it sound to make such a conclusion? Clearly, any external process (that is, any process not involved in the feature or the property) can only affect the behaviour of any of the processes involved in the feature or property via communication to or from such a process. Therefore, as long as all possibilities of such communication is considered, the (internal) behaviour of the external processes does not affect the truth (or otherwise) of the property. The following theorem (stated here without proof) clarifies this reasoning.

**Generalisation Theorem** Suppose that \( N_n = N_n(p_0, p_1, \ldots, p_{n-1}) \) is a network of \( n \) processes such that, for \( m \leq i \leq n - 1 \), \( p_i \) has no features enabled. If \( M(N_m || Abs) \models \phi[0, 1, \ldots, t] \) where \( M(N_m || Abs) \) is the finite-state model representing \( N_m \) acting concurrently with the abstract process \( Abs \) (defined below), \( \phi[0, 1, \ldots, t] \) is a temporal logic formula containing free variables indexed by \( 0, 1, \ldots, t \), where \( 0 \leq t \leq m - 1 \), then \( M(N_n) \models \phi[0, 1, \ldots, t] \).

The consequences of the theorem are the technique outlined in Figure 1 (section 1), where we represent the behaviour of \( p_m || \ldots || p_{n-1} \) by an abstract process, \( Abs \). \( Abs \) can only affect the behaviour of the \( m \) concrete processes through communication. Therefore communication to/from a concrete process from/to any other process takes place via a virtual channel. Rather than concrete processes reading/writing to this (virtual) channel and behaving accordingly, each possible read is replaced by a non-deterministic choice over the possible contents of such a channel. In this way all possible behaviours are explored. (A write to such a channel is no longer relevant.)

A model of the \( m \) concrete processes \( p_0, p_1, \ldots, p_{m-1} \) together with the abstract process is generated automatically from a model of the concrete processes together with a single basic call process \( p \). For example if \( m = 3 \), a model of \( p_0 || p_1 || p_2 || Abs \) is generated from a model of \( p_0 || p_1 || p_2 || p \). In this case, for any \( 0 \leq t \leq 2 \) we show, by model-checking, that \( M(p_0 || p_1 || p_2 || Abs) \models \phi[0, \ldots, t] \) and can hence infer that \( \forall n. M(p_0 || p_1 || p_2 || \ldots || p_{n-1}) \models \phi[0, \ldots, t] \).

We refer to the \( m \) process model plus abstract process model as the \( N \)-users models and in the following section, we describe its generation.
5  The N-users model

The $N$-users model, is generated in the following way: generate the (relevant) model of $m + 1$ processes (the $(m + 1)$-users model) and then convert this model to the $N$-users model. Both steps are done via Perl scripts.

In order to describe the conversion more easily, we will assume that $m = 3$. In this case, the $N$-users model contains a global constant $m$, declared to be equal to 3 and a new global channel $out\_channel$. The $Abs$ process is defined as follows:

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

and initiated within the init process thus:

```plaintext
run Abs(out\_channel)
```

The remaining (concrete) processes are declared in the usual way and communication between the concrete processes is unchanged. To handle the sending or receiving of messages to or from an abstract process, a series of (inline) functions are used to replace polling, writing, rewriting and the checking of channels. An additional function ($\text{name\_channel}$) has also been introduced to set the name of the partner channel when an incoming call from an abstract process is received. In all cases, when an abstract process is not involved, polling, writing, rewriting etc. proceeds as normal.

A call from an abstract process to a concrete process is initiated by a message ($out\_channel$, 0) being placed (by the Abs process) onto the concrete process’s channel. The concrete process now knows that it has a call from an external process and its partner is set to the (local) value $m\_channel$, via the $\text{name\_channel}$ function. (Note that $m\_channel$ is a channel name, whereas $m$ is a constant.) Henceforth any polling of the partner channel is replaced by a non-deterministic choice of all messages that it could contain, via the $\text{poll\_partner}$ function. Similarly any checks to determine whether the partner’s channel is full or empty is replaced by a non-deterministic choice over the value of a new variable $\text{len\_partner}$, which can equal 0 or 1, via the $\text{set\_channel}$ function. Confirmation of a connection (as terminating party) is no longer relevant (there is no concrete partner to check such a confirmation). So, via the $\text{rewrite\_to\_confirm}$ function, no action is taken in this case.
A call from a concrete process to an abstract process is initiated from the dialing state in which the value of partner (a channel name) is set to m_channel. That is the following choice is made:

:: partner[selid] = m_channel; partnerid=m

During a call to an abstract process, a write (alert) to the abstract process’s channel is no longer relevant, the call_partner function ensures that no action is taken in this case. A poll of the (concrete) process’s own channel is replaced by a non-deterministic choice of all messages that it could contain, via the poll_self function. Changing the contents of the partner’s channel, to close down connection (as originating party) is no longer relevant (there is no concrete partner to check such a close down). So, via the rewrite_to_finish function, no action is taken in this case.

When a user prescribes to a ringback feature, and a ringback has been requested, a poll of the channel associated with the user to whom a ringback is to be made is required (via the feature_lookup inline, from the idle state). This gives rise to several problems. Suppose that id1 is the id of the user to whom a ringback is to be made. If id1 = m it is not possible to convert from the id to a channel name via the global chan_name array, as the associated channel in this case is m_channel, a local channel. Also, if id1 = m we can not poll m_channel (the poll_partner inline can not be used as the partner has not yet been set, and also requires the channel name as input rather than an id). To avoid the first problem, in the N-user model we no longer use the chan_name array, but instead introduce an inline function convert_id_to_channel which sets a new local variable chan_name to zero, one etc. when id1 = 0, 1 etc. and to m_channel if id1 = m. To avoid the second problem, an inline function set_other_channel is used, which either polls the channel associated with id1 (if id1 ≠ m) or, if id1 = m, non-deterministically assigns a value to a new local variable len_other, which can equal 0 or 1.

The RWF feature presents another problem. It is feasible that an abstract process could request a ringback from a (concrete) user that prescribes to this feature. Therefore, this must be reflected in the N-user model in this case (all ways in which an abstract process can “interfere” with a concrete process must be reflected in the N-process model). Hence, if User[i] has RWF, the Abs process may, as well as writing to the channels of any process, set the value of return[i] to m. Thus, if User[i] has RWF, the Abs proctype becomes

11
```c
proctype Abs (chan self)
{
  do :: zero!self, 0
  :: one!self, 0
  :: two!self, 0
  :: return[1]=m
  od}
```

Notice that this last choice is not included in the example below, as no RWF feature is enabled in this case.

All arrays, apart from the feature arrays, now have size equal to the number of concrete processes ($m$). The feature arrays have size $m+1$. Setting $CFU[m] = 0$, for example, denotes that no abstract users prescribe to the call forwarding feature. (Indeed, all such array elements will be equal to zero, as, by definition, no abstract processes prescribe to features.)

The full Perl script, to convert any 4-user model, with one feature, to an $N$-user model, is given in the appendix.

Below we give the Promela code for the $N$-user model in the case where there is one feature ($User[1]$ forwards calls unconditionally to $User[2]$) together with an associated property to be verified (property 1, described in section 2.2). Note the addition of the inline functions and the new $Abs$ proc type. For ease of understanding, all code within the user process and the `feature` lookup inline that is different from that in the original 4-user process with $CFU[1] = 2$ is printed within a rectangular box. Note that we are no longer able to make assertions about the contents of channels, as we have allowed greater choice in the types of messages that are considered (to and from the abstract processes). Thus such assertions have been commented out of the code (but indented, to distinguish them from editorial comment).

```c
stype = {on, off, dial, call, ringing, tring, unobt, engaged, connected, disconnect, ringbacker,
  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 out_channel = [1] of {chan.bit};
chan partner[3];

/* additional declarations:*/
byte CFU[4];
stype dev[3] = on;
byte dialed[3] = 6;
short p0=-1;
#define m 3;
```
inline feature_lookup (q1, id1, st)
{
    do
        ::(((id1-7)&(st—and_dial))&CFU[id1]]-6)) ->
        id1=CFU[id1];
        convert_id_to_channel[m_channel,id1,chan_name]; q1 = chan_name;
        ::else->break
    od
}

inline poll_partner(q)
{if
    ::(partner[selfid]==q)->
    if
        ::messchan=self; messbit=0
        ::messchan=mask; messbit=1
        ::messchan=all; messbit=0
        ::messchan=mask; messbit=1
    fi
    :: else -> partner[selfid] & messchan & messbit
    fi}

inline poll_self(q)
{if
    ::(partner[selfid]==q)->
    if
        ::messbit=0
        ::messbit=1
    fi
    :: else->self & messchan & messbit
    fi}

inline call_partner(q)
{if
    :: (partner[selfid]!=q)->
        partner[selfid]! self,0;
    :: else -> skip
    fi
}

inline rewrite_to_finish(q)
{if
    ::(partner[selfid]!=q)->
        partner[selfid]& messchan & messbit;
        partner[selfid]! messchan,0
    ::else->skip
    fi}

inline rewrite_to_confirm(q)
{if
    ::(partner[selfid]!=q)->
        partner[selfid] & messchan & messbit;
        partner[selfid]! self,1
    :: else -> skip
}
inline name_channel(q)
{
  :: (partner[selfid]==out_channel)->
    partner[selfid]=q;
    self->messchan.messbit;
    self|partner[selfid].messbit
  :: else->skip
  fi
}

inline set_partner_channel(q)
{
  :: (partner[selfid]==q)->
    if
      :: len_partner==0
      :: len_partner=1
    fi
    :: else -> len_partner=len(partner[selfid])
  fi
}

inline set_other_channel(channelid)
{
  :: (channelid==0)->len_other=len(zero)
  :: (channelid==1)->len_other=len(one)
  :: (channelid==2)->len_other=len(two)
  :: else ->
    if
      :: len_other==0
      :: len_other=1
    fi
  fi
}

inline convert_id_to_channel(q,qid,q2)
{
  :: (qid==0)->q2=zero
  :: (qid==1)->q2=one
  :: (qid==2)->q2=two
  :: else ->q2=q
  fi
}

/* no event inline required */
/* no network_event inline required */

proctype User (byte selfid; chan self)
{
  chan messchan=null;
  chan m_channel=null;
  bit messbit=0;
  bit len_partner=0;
  bit len_other=0;
  stype state=on;
  byte partnerid=6;
  idle:
  atomic
  {assert(dev[selfid]==on); assert(partner[selfid]==null);
    assert(state==on); assert(partnerid==6);
/* either attempt a call, or receive one */
if
  :: empty(self) -> state=st_idle;
  feature_lookup(partner[selfid], partnerid, state);
  if
    :: elm->state=on
    fi;
  dev[selfid]=off; self!self,0; goto dialing
/* no connection is being attempted, go offhook */
/* and become originating party */
  :: (len(self)!=1)->self!partner[partnerid], mesbit T;
  /* an incoming call */
name_channel(m_channel); set_partner_channel(m_channel);
if
  :: [len_partner == 1] T > len_partner == 0; poll_partner(m_channel);
if
  :: meschan = self TG call attempt still there */
    meschan.null; mesbit=0; goto talert
  else -> self?meschan, mesbit; /* call attempt cancelled */
    partner[selfid]=null; partnerid=6; meschan=null; mesbit=0;
    goto preidle
fi
:: [len_partner == 0] T >
  self?meschan, mesbit; /* call attempt cancelled */
  partner[selfid]=null; partnerid=6; meschan=null; mesbit=0;
  goto preidle
fi);
dialing;
atomic
{assert(dev[selfid]=off);
 assert(full(self));
 assert(partner[selfid]=null);
/* dial or go onhook */
if
  ::
/* 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
      :: partnerid= 7;
    fi;
    state=st_dial;
    feature_lookup(partner[selfid], partnerid, state);
    if
      :: state=st_unobt -> state=on; partner[selfid]=null; partnerid=6;
        dial[6]=6; goto unobtainable
      :: (state=st_dial&partnerid!=7) -> state=on; goto calling
      :: (state=st_dial&partnerid!=7) -> state=on; partner[selfid]=null;
        partnerid=6; dial[6]=6; goto unobtainable
      :: (state=st_redial) -> state=on; partnerid=6;
dialled[selfid]=6; goto dialing
fi
::dev[selfid]=on; self?messchan,messbit;
/*assert(messchan==self);*/
messchan=null; messbit=0; goto preidle
/*go onhook, without dialing */
fi);

calling: /* check number called and process */
atomic
assert(dev[selfid]==off);
assert(full[selfid]);
state=st_call2;
feature_lookup[partner[selfid],partnerid,state);
if
::state=st_unobt->state=on; partner[selfid]=null; partnerid=6;
dialled[selfid]=6; goto unobtainable
::state=st_call2->state=on; skip
fi;
if
:: partner[selfid]==self -> goto busy
/* invalid partner */
:: partner[selfid]!=self ->

set_partner_channel(m_channel);
if
:: (len_partner==1)->

len_partner=0;

/* valid partner but engaged */
:: (len_partner==0)->

all_partner(m_channel)

self?messchan,messbit; self!partner[selfid],0;
messchan=null; messbit=0; goto callert
/* valid partner, write token to partner's channel */
fi
fi);

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

/* trivial dial *//
fi);

/* comment out entire ringback state when no ringback feature switched on */

/* ringback state not required */

unobtainable:
/* number called is unobtainable, go onhook or trivial dial */
atomic {
  assert(full(self));
  assert(partner[s elfid]—null);
  assert(partnerid=0);
  if :
dev[ selfid]—on; self? messchan, messbit;
  / assert (messchan—self):*/
    messchan—null; messbit=0; goto preidle
/* go onhook, cancel connection attempt */
/* : event_action(dial); goto busy */
/* trivial dial */
  fi};

calret:
/* called party is ringing */
atomic {
  /* assert((len(partner[ selfid])—1));*/
    assert(full(self));
    assert(dev[ selfid]—off);
    poll_self[m, channel];
  / assert (messchan= partner[ selfid]);*/
    messchan=null;
  /* check channel */
    if :
      messbit=1—messbit=0; goto oconnected
  /* correct token */
      messbit=0—goto calret
  /* wrong token, not connected yet, try again */
      messbit=0—goto origout
  /* give up */
      /* : event_action(dial); messbit=0; goto calret*/
  /* trivial dial */
      fi};

origout: /* abandon call attempt */
atomic {
  /* assert((len(partner[ selfid])—1));*/
    assert(full(self));
    assert(dev[ selfid]—off);
    dev[ selfid]—on; self? messchan, messbit;
    rewrite_fin(channel);
    partner[ selfid]—null; partnerid=0; dialed[ selfid]—6; messchan—null; messbit=0; goto preidle;
  /* give up, go omh hook */
};

17
oconnected:
  atomic
  {assert(full(self));
   /*assert([len[partner[selfid]]-1]); */
  
  /* connection established */
  goto oclose;

oclclose: /* disconnect call */
  atomic
  {assert(full(self));
   /*assert([len[partner[selfid]]-1]); */
     dev[selfid]=on; self? messchan.messbit;
  
  /* empty own channel */
     /assert(messchan= partner[selfid]);*/
  /assert(messbit--1); /*
     rewrite_fo_finish(m_channel);
  */

  /* empty partner's channel */
  /assert(messchan= self); assert(messbit--1);*/
  /* and disconnect partner */
     partner[selfid]=null; dialled[selfid]=6; partnerid=6;
     messchan=null; messbit=0; goto preidle;

talert:
  atomic
  {assert(dev[selfid]=on); assert(full(self));
  
  /* either device rings or*/
  /* connection attempt is cancelled and then empty channel */
  poll_partner(m_channel);

if
  :: messchan=self->messchan=null; messbit=0; goto tpickup
  else->skip /* attempt has been cancelled */
fi;
self? messchan, messbit; partner[selfid]=null; partnerid=6;
  dialled[selfid]=6; messchan=null; messbit=0; goto preidle;

  tpickup:
  /* proceed with connection or connect attempt cancelled */
  atomic
  {assert(full(self));

  sel_partner_channel(m_channel);

  if
  :: (len_partner ==1) -> len_partner = 0;
  poll_partner(m_channel);

  if
  :: messchan=self ->
  /*connection proceeding */
  /*assert(messbit =0); */
  self? messchan, messbit;
     /assert(messchan= partner[selfid]); assert(messbit--0);*/
  dev[selfid]=off;
     rewrite_fo_con_firm(m_channel);

18
/* establish connection */
    self!partner[selfid].1;
    messchan=null; messbit-0; goto tclose
  :else -> self!messchan.messbit;
/* wrong message, connection cancelled */
    dev[selfid]=on; partner[selfid]=null; dialed[selfid]-6;
    partnerid-6; messchan=null; messbit-0; goto preidle
  fi
  :else -> self!messchan.messbit;
/* connection cancelled */
    dev[selfid]=on; partner[selfid]=null; partnerid-6;
    dialed[selfid]-6; messchan=null; messbit-0; goto preidle
  fi;

  tclose:
  /* check if originator has terminated call */
  atomic
  {
    self!df(m_channel);
    if
      ::(messbit == 1 && dev[selfid]==off) ->
/* trivial handset down */
      dev[selfid]=off; messchan=null; messbit-0; goto tclose
  ::(messbit == 1 && dev[selfid]==on) ->
/* trivial handset up */
      dev[selfid]=on; messchan=null; messbit-0; goto tclose
  ::(messbit == 0 && dev[selfid]==off) ->
/* disconnect tone */
      dev[selfid]=on;
  /* connection is terminated */
      self!messchan.messbit; partner[selfid]=null; partnerid-6;
      dialed[selfid]-6; messchan=null; messbit-0;
      goto preidle
  fi;
  preidle:
  atomic
  {goto idle}
}
} /* end User */

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

/*Clam7*/
#define p ((dialed[0]==1) && (User[p0]@calling))
#define q (dev[0]==on)
#define r ((partner[0]==two)&&(User[p0].alert)|| (User[p0].busy))

init
{
    atomic
    {partner[0]=null;
     partner[1]=null;
     partner[2]=null;

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

     p0= run User(0,zero);
     run User(1,one);
     run User(2,two);
     run Abs(out_channel);
    }
}
}

#include "claim7"

6 Verification results

6.1 Feature validation

Each feature is validated by verifying that the associated property holds for all models in which that feature (only) is enabled. The following tables compare (detailed) verification results for each of the features and corresponding properties described in [5]. In each case we compare results for the 3-user case, the 4-user case and the N-user case. In order to be consistent with earlier results [5], the properties are numbered from 7 to 15(b). Property 7 is the property associated with the CFU feature, properties 8a and 8b are those associated with CFB, and the remaining properties are those associated with OCS, ODS, TCS, RBWF, OCO, TCO and RWF (15(a) and 15(b)). Note

**Depth** describes the length of the longest path (×10^6) explored during the search

20
Table 1: (a) Validation Results – 3 users

<table>
<thead>
<tr>
<th>Property</th>
<th>Depth</th>
<th>States</th>
<th>Mem</th>
<th>Time</th>
<th>State-vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0.09</td>
<td>0.07</td>
<td>0.3</td>
<td>2.8</td>
<td>96</td>
</tr>
<tr>
<td>8a</td>
<td>0.2</td>
<td>0.2</td>
<td>0.6</td>
<td>3.8</td>
<td>96</td>
</tr>
<tr>
<td>8b</td>
<td>0.2</td>
<td>0.1</td>
<td>0.5</td>
<td>3.5</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>0.1</td>
<td>0.08</td>
<td>0.3</td>
<td>2.9</td>
<td>108</td>
</tr>
<tr>
<td>10</td>
<td>0.1</td>
<td>0.07</td>
<td>0.3</td>
<td>2.9</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>0.1</td>
<td>0.08</td>
<td>0.3</td>
<td>2.8</td>
<td>108</td>
</tr>
<tr>
<td>12</td>
<td>0.7</td>
<td>0.4</td>
<td>1.8</td>
<td>7.6</td>
<td>108</td>
</tr>
<tr>
<td>13</td>
<td>0.07</td>
<td>0.05</td>
<td>0.2</td>
<td>2.6</td>
<td>108</td>
</tr>
<tr>
<td>14</td>
<td>0.06</td>
<td>0.04</td>
<td>0.2</td>
<td>2.4</td>
<td>108</td>
</tr>
<tr>
<td>15(a)</td>
<td>0.8</td>
<td>0.5</td>
<td>2.0</td>
<td>8.3</td>
<td>108</td>
</tr>
<tr>
<td>15(b)</td>
<td>0.8</td>
<td>0.5</td>
<td>2.2</td>
<td>8.9</td>
<td>108</td>
</tr>
</tbody>
</table>

**States** is the number of states stored ($\times 10^5$)

**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.

The reason that the state-space is so much smaller in the N-user case than the 4-user case is that, unlike a physical fourth process, the Abs process has no local states (channels from the abstract processes do not actually exist for example). The N-user process model is, by comparison to a 4-user model, very economical.

These results prove not only that the approach is tractable, but it is *sound*. Namely, the Abs process is an abstraction of the $n - m$ processes because we can not observe any *behavioural* difference between the N-user model and an $n$-user model, for any $n$. (Clearly we do not have validation results for an $n$-user model for $n \geq 4$, but know how any $n$-user model is *expected* to behave). Comparing the validation results for the N-user model and the 4-user model, the results are unchanged, only the time taken to prove a result changes.
Table 2: (b) Validation Results – 4 users

<table>
<thead>
<tr>
<th>Property</th>
<th>Depth</th>
<th>States</th>
<th>Mem</th>
<th>Time</th>
<th>State-vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7.0</td>
<td>4.2</td>
<td>16.1</td>
<td>80</td>
<td>116</td>
</tr>
<tr>
<td>8a</td>
<td>16.0</td>
<td>12</td>
<td>42.8</td>
<td>204</td>
<td>116</td>
</tr>
<tr>
<td>8b</td>
<td>16.0</td>
<td>10</td>
<td>38.3</td>
<td>180</td>
<td>116</td>
</tr>
<tr>
<td>9</td>
<td>11.0</td>
<td>6.4</td>
<td>22.7</td>
<td>129</td>
<td>140</td>
</tr>
<tr>
<td>10</td>
<td>10.0</td>
<td>6.1</td>
<td>21.7</td>
<td>111</td>
<td>128</td>
</tr>
<tr>
<td>11</td>
<td>11.0</td>
<td>6.3</td>
<td>22.4</td>
<td>128</td>
<td>140</td>
</tr>
<tr>
<td>12</td>
<td>82.0</td>
<td>46</td>
<td>170.1</td>
<td>900</td>
<td>132</td>
</tr>
<tr>
<td>13</td>
<td>6.0</td>
<td>3.2</td>
<td>11.6</td>
<td>50</td>
<td>140</td>
</tr>
<tr>
<td>14</td>
<td>5.0</td>
<td>3.2</td>
<td>11.5</td>
<td>61</td>
<td>140</td>
</tr>
<tr>
<td>15(a)</td>
<td>96.0</td>
<td>51</td>
<td>193.0</td>
<td>1121</td>
<td>132</td>
</tr>
<tr>
<td>15(b)</td>
<td>97.0</td>
<td>54</td>
<td>202.2</td>
<td>1120</td>
<td>132</td>
</tr>
</tbody>
</table>

Table 3: (c) Validation Results – N users

<table>
<thead>
<tr>
<th>Property</th>
<th>Depth</th>
<th>States</th>
<th>Mem</th>
<th>Time</th>
<th>State-vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2.0</td>
<td>0.8</td>
<td>3.2</td>
<td>17.5</td>
<td>116</td>
</tr>
<tr>
<td>8a</td>
<td>2.0</td>
<td>1.5</td>
<td>5.3</td>
<td>28.7</td>
<td>116</td>
</tr>
<tr>
<td>8b</td>
<td>2.0</td>
<td>1.2</td>
<td>4.8</td>
<td>25.5</td>
<td>116</td>
</tr>
<tr>
<td>9</td>
<td>2.0</td>
<td>0.9</td>
<td>3.4</td>
<td>18.8</td>
<td>124</td>
</tr>
<tr>
<td>10</td>
<td>2.0</td>
<td>0.8</td>
<td>3.3</td>
<td>17.9</td>
<td>112</td>
</tr>
<tr>
<td>11</td>
<td>2.0</td>
<td>0.9</td>
<td>3.4</td>
<td>18.6</td>
<td>124</td>
</tr>
<tr>
<td>12</td>
<td>16.0</td>
<td>6.4</td>
<td>25.0</td>
<td>128.9</td>
<td>116</td>
</tr>
<tr>
<td>13</td>
<td>1.0</td>
<td>0.7</td>
<td>2.6</td>
<td>15.3</td>
<td>124</td>
</tr>
<tr>
<td>14</td>
<td>1.0</td>
<td>0.4</td>
<td>1.6</td>
<td>10.2</td>
<td>124</td>
</tr>
<tr>
<td>15(a)</td>
<td>16.7</td>
<td>7.2</td>
<td>27.4</td>
<td>158.3</td>
<td>116</td>
</tr>
<tr>
<td>15(b)</td>
<td>18.6</td>
<td>8.0</td>
<td>30.4</td>
<td>170.4</td>
<td>116</td>
</tr>
</tbody>
</table>
6.2 Pairwise analysis of features

This is the ultimate aim of this work. We give here a simple example (where the number of concrete users required is 3). Here the features present are $CFU[1] = 2$, $CFU[1] = 0$ and the property is property 1 (see section 2.2) with $j = 1$, $k = 2$, $i = 0$, thus:

*If user 0 rings user 1 then a connection between 0 and 2 will be attempted before user 0 hangs up.*

We have already shown [5] that a feature interaction exists here: the property holds when the first feature is enabled alone, but not when both features are enabled (if the second feature takes precedence, the attempted call from user 0 will be forwarded to user 0, not to user 2). This scenario is offered as a counter-example during a SPIN verification of the 4-user model. The same scenario is offered during verification of the corresponding $N$-user model.

We have yet to prove any similar results for when the number of concrete users is greater than 3 (for example to examine whether the same property holds for $n$ users when the second feature above is replaced by $CFU[3] = 4$ would require 5 concrete users). This is current work.

6.3 The number of concrete processes

For each pair of features and property to be verified, the number of concrete processes is equal to the number of (distinct) variables occurring in the features and property. Suppose the features are $f_1$ and $f_2$ and the property to be verified is $\phi_{f_1}$ – the property relating to $f_1$.

For our set of features, each feature has at most 2 variables and the properties have at most 1 variable that is not contained within the associated feature description. Hence the maximum number of distinct variables that is possible is 5, (2 each for $f_1$ and $f_2$ and one further variable for $\phi_{f_1}$). The example above ($f_1 = CFU[1] = 2$, $f_2 = CFU[3] = 4$, with property 1 ($j = 1$, $k = 2$ and $i = 0$) is such a case.
7 Discussion

The results above clearly demonstrate the feasibility of the abstraction technique for this application domain – the model checking requirements are well within the capability of our machine. Also, the transformation to a N-user model is relatively straightforward: we need only consider the communication between the external processes and the concrete processes. On the other hand, an induction approach [8, 15] requires the construction of an inductive invariant. This is a non-trivial exercise as it involves incorporating the behaviour of the entire system within the invariant.

We have successfully applied the abstraction approach in the feature interaction analysis domain. We are unaware of any previous generalisation results in this domain.

8 Conclusions

The automatic verification of concurrent systems by model-checking has traditionally been limited due to the inability to generalise results to systems consisting of any number of processes. This may be a serious limitation because it is often important to show that results do, or do not, scale up.

In this paper we show a technique that can be used to prove general results about a concurrent system of an arbitrary number of processes. The technique does not involve explicit induction, and consequently is rather simpler to apply. The key idea is to consider a system of constant number \( m \) of concurrent processes, in parallel with one abstract process which represents the product of any number of other processes.

We have applied the technique to feature interaction analysis of a telecommunications service. The initial model on \( m \) processes (for any specified set of selected features) is automatically generated from a template, using Perl scripts. Similarly, the conversion from an \( m \) process model to an \( m \) process plus abstract model, representing an arbitrary \( n \) process model, is performed automatically. We give empirical results, which both demonstrate the feasibility of the approach and show that our interaction results scale up.
Acknowledgments

The authors would like to thank Ken McMillan for valuable discussions relating to this work.

References


Appendix

We include here the full Perl script to convert a 4-user model, with one feature enabled, to an N-user model.

26
#!/usr/local/bin/perl
# creates an abstract model, Abs.p from a four user model, model.p
$n_o_procs=4;
# open files for template and destination
open(TEMPLATE, "model.p");
open(DCHECK, "Abs.p");

# write the inline functions

$pollp_inline="inline poll_partner(q)
\{if
  :: (partner[selfid] == q) ->
  if
    :: messchan=self; messbit=0
    :: messchan=self; messbit=1
    :: messchan=null; messbit=0
    :: messchan=null; messbit=1
  fi
  :: else -> partner[selfid]?messchan,messbit>
  fi

\}\n";

$polls_inline="inline poll_self(q)
\{if
  :: (partner[selfid] == q) ->
  if
    :: messbit=0
    :: messbit=1
  fi
  :: else -> self?messchan,messbit>
  fi

\}\n";

$callp_inline="inline call_partner(q)
\{if
  :: (partner[selfid] != q) ->
  partner[selfid]=self,0;
  :: else -> skip
  fi

\}\n";

$rtffin_inline="inline rewrite_to_finish(q)
\{if
  :: (partner[selfid] != q) ->
  partner[selfid]?messchan,messbit;
  partner[selfid]=messchan,0
  :: else -> skip
  fi

\}\n";

$rtcom_inline="inline rewrite_to_confirm(q)
\{if
  :: (partner[selfid] != q) ->
  partner[selfid]?messchan,messbit;
  partner[selfid]=self,1
  :: else -> skip
  fi
```

\n
$\text{setchan inline="inline name\_channel(q)"

\text{\textbackslash if}\n
\text{:: \{partner\_selfid=\_out\_channel\}->\}

\text{partner\_selfid}=q;\n
\text{self\_messchan\_messbit;\n
\text{self\_partner\_selfid\_messbit\n
\text{:: else=\_skip\n
\text{fi\n
\n
$\text{setpchan inline="inline set\_partner\_channel(q)"

\text{\textbackslash if}\n
\text{:: \{partner\_selfid=\_q\}->\}

\text{if}\n
\text{:: len\_partner=0\n
\text{len\_partner=1\n
\text{fi}\n
\text{:: else=\_len\_partner=\_len(partner\_selfid)\n
\text{fi\n
\n
$\text{setchan inline="inline set\_other\_channel(channelid)"

\text{\textbackslash if}\n
\text{:: \{channelid=0\}->\_len\_other=\_len(zero)\n
\text{:: \{channelid=1\}->\_len\_other=\_len(one)\n
\text{:: \{channelid=2\}->\_len\_other=\_len(two)\n
\text{:: else=\}

\text{if}\n
\text{:: len\_other=0\n
\text{len\_other=1\n
\text{fi\n
\n
$\text{convert inline="inline convert\_id\_to\_channel(q,qid,q2)"

\text{\textbackslash if}\n
\text{:: \{qid=0\}->q2=\_zero\n
\text{:: \{qid=1\}->q2=\_one\n
\text{:: \{qid=2\}->q2=\_two\n
\text{:: else=q2=q\n
\text{fi\n
\n
#determine which feature is activated

while (line = <TEMPLATE>){\n
\text{if($line= /a/.\_CFU.\*/agua/{feature="CFU",;\n
\text{if($line= /a/.\_CFB.\*/agua/{feature="CFB",;\n
\text{if($line= /a/.\_CS.\*/agua/{feature="CS",;\n
\text{if($line= /a/.\_GCS.\*/agua/{feature="GCS",;\n
\text{if($line= /a/.\_TCS.\*/agua/{feature="TCS",;\n
\text{if($line= /a/.\_XRS.\*/agua/{feature="XRS",;\n
\text{if($line= /a/.\_XGD.\*/agua/{feature="XGD",;\n
\text{if($line= /a/.\_XTC.\*/agua/{feature="XTC",;\n
\text{if($line= /a/.\_XIF.\*/agua/{feature="XIF",;\n
\text{if($line= /a/.\_XVI.\*/agua/{feature="XVI",;\n
\text{if($line= /a/.\_XTV.\*/agua/{feature="XTV",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
\text{if($line= /a/.\_XRF.\*/agua/{feature="XRF",;\n
```
return($propI)=null;else{$WFAbsline="\n";}

#write the N-users model

$newdescribeLine="\n* N-users model with 1 Feature, $feature \n":
$newAbsChan="ch \nout\nchannel = [1] of \{ch, bit\}: 
"
$newLine="\nbit len\nother=0; \nchannel m\nchannel=\nnull; \ch an chan\n\n_name=\nnull;
"
$newIncoming="\n\n* an incoming call \n/
name\nchannel m\nchannel());
"
$newfull="set\npartner\nchannel(m\nchannel())
"
if
:: (len\npartner--1) \n\rightarrow \nlen\npartner=0;
"
$newempty="": (len\npartner==0) \n\rightarrow 
"
$newLine=""\n#define = 3
"
"\n* The simple basic call protocol: A rings B\n":

$newnewline="$pollp\nnewline\n", "$polls\ninline\n", "$callp\ninline\n",
"$rvtf0\ninline\n", "$rvtco\ninline\n", "$mchannl\ninline\n",
"$stchann\ninline\n", "$stochann\ninline\n", "$convert\ninline\n",
"#proctype User (byte selfid\n: chan_self)"

$newassignChan="convert\n_id_to\nchannel(m\nchannel.id1,chan\n_name)\n",
$q1=chan\n_name;
chan\n_name=\nnull;
"
#The new Abs process

$newAbsProc="\n end User

/\n")

proctype Abs \n(chan self)\n{
do :: zero\nself,0
:: one\nself,0
:: two\nself,0
$WFAbsline
od\n"; close(TEMPLATE);
open(TEMPLATE, "model.p");
while ($line = $TEMPLATE){
if ($line =~ /\*Four users model.*/$newdescribeLine/){
if ($line =~ /\*Generated from Template.*$/){
if ($line =~ /\*chan three.*$/){
if ($line =~ /\*convert from number.*$/){
if ($line =~ /\*The simple basic call protocol.*$/){
if ($line =~ /\*proctype User.*$/){
if ($line =~ /\*chann\n messchan=\nnull.*$/){
if ($line =~ /\*an incoming call.*$/){
if ($line =~ /\*len\npartner=\selfid\n\} \n\rightarrow \newemptyp});
if ($line =~ /\*\len\n\channel\n[rgbknum=\selfid\n] \n\} \n\rightarrow \n/){
:: (len\n\other--1) \n\rightarrow \nlen\n\other=0;
}
if ($line =~ /\*\len\n\\channel\n[rgbknum=\selfid\n] \n\} \n\rightarrow \n/){
:: (len\n\other--1) \n\rightarrow \nlen\n\other=0;
}
```c
::(len_other==1)\rightarrow len_other=0;\}\nif (\$line== a::\{(len\chan_name\return[\selfid\]}\})-1.*/\n::(len_other==0)\rightarrow st_rback1();\}
if (\$line== a::\{(len\chan_name\return[\selfid\]}\})-1.*/\n::(len_other==1)\rightarrow len_other=0;\}\nif (\$line== a::*partner-polling.*\poll_partner(m\channel)\}/)\{
if (\$line== a::*self-polling.*\poll_self(m\channel)\}/)\{
if (\$line== a::*calling-partner.*\call_partner(m\channel)\}/)\{
if (\$line== a::*finish-call.*\rewrite_to_finish(m\channel)\}/)\{
if (\$line== a::*startback-in-idle-line.*\set_other_channel(\rgbknum[\selfid\})\}/)\{
if (\$line== a::*reset-in-idle-line.*\set_other_channel(\rgbknum[\selfid\})\}/)\{
if (\$line== a::*confire-call.*\rewrite_to_confirm(m\channel)\}/)\{
if (\$line== a::*run User\(3.*\run Abs(out\channel)\}/)\{
if (\$line== a::*partner[\selfid\}) - three.*\partner[\selfid\}) = m\channel;\}/)\{
if (\$line== a::*dalled[\selfid\})=3.*\dalled[\selfid\})=m\channel;\}/)\{
if (\$line== a::*\q1-chan_name[\lid1]. */\$meassign\chan)\}/
if (\$line== a::*\partner[\3])=.*\)/
if (\$line== a::*\chan_name[\6])=.*/\)
if (\$line== a::*\partnerid[\3])=.*\)
if (\$line== a::*\send User.*\$meAbsproc\}
if (\$line== a::*\chan_name[\0])=zero.*\)/
if (\$line== a::*\chan_name[\1])=one.*\)/
if (\$line== a::*\chan_name[\2])=two.*\)/
if (\$line== a::*\chan_name[\3])=three.*\)/
if (\$line== a::*\chan_partner[\4])=.*\chan_partner[\3]\}/\)
if (\$line== a::*\type dev[\4])=.*\type dev[\3]\=}on\);\)/
if (\$line== a::*\type network\event[\4])=.*\type network\event[\3]\=}on\);\)/
if (\$line== a::*\type event[\4])=.*\type event[\3]\=}on\);\)/
if (\$line== a::*\byte rgbknum[\4])=.*\byte rgbknum[\3]\=}6\);\)/
if (\$line== a::*\byte retnum[\4])=.*\byte retnum[\3]\=}6\);\)/
if (\$line== a::*\byte dalled[\4])=.*\byte dalled[\3]\=}6\);\)/
if (\$line== a::*\assert(messchan==self).*/\n\*/\assert(messchan==self);\}/\)
if (\$line== a::*\assert(messchan==partner).*/\n\*/\assert(messchan==partner[\selfid\})\}/\)
if (\$line== a::*\assert(messchan==partner[\selfid\})\)/\)
if (\$line== a::*\assert(messbox=1)\:. */\assert(messbox-1);\}/\)
if (\$line== a::*\assert(messbit=0). */\assert(messbit=0);\}/\)
if (\$line== a::*\assert(full\partner. */\assert(\{\{len(\partner[\selfid\})=1);\}/\)
print \$MODEL "$\line$";}
#close the files
close(\TEMPLATE); close(\MODEL);
```