A Feature Construct for PROMELA*

Malte Plath and Mark Ryan
School of Computer Science
University of Birmingham
Birmingham B15 2TT
UK
`http://www.cs.bham.ac.uk/{~mcp,~mdr}`

August 1998

Abstract

It is common practice in software development to extend the lifespan of products by enhancing and upgrading them with new functionality, or "features". This is seen as a cheaper, quicker and easier solution than redeveloping the system from scratch. However, a new feature will usually override previous behaviours of the system and thereby inadvertently introduce inconsistencies, i.e., bugs. Detecting and avoiding such inconsistencies is, for example, part of what has become known as the "feature interaction problem" in the telecommunications industry.

To facilitate the detection of inconsistencies between basic system and features, using model checking, we extend PROMELA with a *Feature Construct*. This construct allows us to define changes to a given PROMELA program in a clearly structured and intuitive way. The features thus defined are then automatically integrated into the PROMELA description of the base system by means of a preprocessor. The developer need not (and should not) alter the original code except through the feature integration process.

1 Introduction

The concept of *features* has emerged in telephone systems analysis as a way of describing optional services to which telephone users may subscribe.

*Financial support from the EU through Esprit working groups ASPIRE (22704) and FIREworks (23531), and from British Telecom and the Nuffield Foundation in the UK is gratefully acknowledged.

Features offered by telephone companies include call-forwarding, automatic-call-back, and voice-mail. We see features as a more general notion, though. Any part or aspect of a specification which the user perceives as having a self-contained functional role is a feature. For example, a printer may exhibit such features as: ability to understand PostScript; Ethernet card; ability to print double-sided; having a serial interface; and others.

Thinking in terms of features is important to the user, who often understands a complex system as a basic system plus a number of features. It is also an increasingly common way of designing products.

To support this way of building a system from a basic system by successively adding features, we have developed a *feature construct* for PROMELA. This allows us to specify features separately from the system and integrate them into the PROMELA description of the system by means of a preprocessor or *feature integrator*. The main aim of our approach of extending a specification and verification language with a feature construct is to provide a 'plug-and-play' system for experimenting with features.

In [1], Cassez has taken a similar approach, but his features can merely manipulate the communication between processes. With our Feature Construct it is also possible to change the computations within a process. Our notion of features is related to *superimpositions* [2], and the feature construct can be seen as a concretisation of the construct which Katz proposes in his paper [2].

Like Katz' approach, our approach is quite general, and is not tied to any particular system description language: we have already developed a feature construct and the corresponding feature integrator for the SMV model checking system.

The structure of our paper is as follows. After a short general description of the concept of a feature for a PROMELA program we give a detailed account of the feature construct and of how features are integrated into a program. This is followed by some examples. Finally we give a summary of our experiences and point out some future work.

2 Features for Promela

A PROMELA system consists of one or more processes, which can communicate through buffered or synchronous channels. Each process is an instance of a process definition (`proctype`), each of which may be instantiated any number of times.

A feature usually changes one or more process definitions; it can introduce new global and local variables, and it can add new statements to a definition or change existing ones. It is also possible to define completely new processes in a feature, or to create further instantiations of existing process definitions. Maybe the most important aspect is that a feature can change the communication structure, i.e. it may divert certain messages or whole channels.

A nice example is given by Katz [2]: we can add to an existing system a monitoring process which collects statistical data from the processes, e.g. how many messages they have sent or how often a certain piece of code has been executed. To accomplish this we not only have to add the monitoring process but we also must make the other processes send the relevant information to the monitor.

Other examples are readily found in telephone systems, where, e.g. the "Voice Mail on Busy" service would be appropriately modelled by a new process for the voice mail recorder. The phone processes (or possibly the process modelling the exchange) would then have to be modified so that they do not return a busy tone to a caller but instead divert her call to the recorder.

3 The Feature Construct for Promela

In this section we describe in some detail the syntax of the feature construct and what the different clauses mean when a feature is integrated into a system.

For the purpose of feature integration we make the additional assumption that every process has at most one infinite loop, to which we refer as the *main loop* from now on. Thus every process either terminates after one pass through it's sequence of statements, or it enters the main loop after some initialisation code and only leaves it, if at all, to terminate execution. The rationale behind this assumption is discussed in section 3.2.1.

3.1 Syntax of the Feature Construct

A feature consists of a `feature` declaration, one or more `roletype` declarations and optionally new `proctype` definitions.

3.1.1 The 'feature' declaration.

The `feature` declaration (Fig. 1) names the feature, declares new global variables, and it states which `roletypes` apply to which `proctype` defini-

```
feature name
{
 [ uses global variables ]

 [ new global variables ]

 { apply roletype-name(parameters) to proctype-name; }*

 { run proctype-name (parameters); }*
}
```

Figure 1: The `feature` declaration

tions. The `feature` section may also require certain global variables to be present in the system and it can instantiate processes.

The `apply` clause is the centerpiece of the feature: it tells the feature integrator which `proctype` declarations should be modified according to which role-type. The actual changes are then given in the role-type definition.

Currently variables are bound simply by name, i.e. they are "captured". The original variables from the `proctype` definition, that a feature uses, must be explicitly named in the feature, the specifier must take care to choose new names for new variables. Functions (preprocessor macros and Promela inline procedures) are declared like variables.

We plan to implement full parametrisation and variable binding as Katz proposes in [2]; this would then replace the `uses` clause in the `roletypes` (see section 3.1.3). However, this will require renaming of global variables to avoid name conflicts, but at the same time we want to retain the original names for subsequent feature integrations.

3.1.2 The 'roletype' declarations.

A role-type declaration (Fig. 2) defines the actual changes to carry out. In the `uses` clause it may demand that certain local and global variables be in scope in the `proctypes` it is applied to. (Again functions are declared like variables.) In the final version, all existing variables that a role-type requires will be listed as parameters; they will then be bound in the `apply` clause in the `feature` declaration. A role-type can declare new local variables (including their initial values), and its structure offers various ways to add

```
roletype name(parameters)
{
   [ uses variables ]
   [ new local variables ]

   [ initial code ]
   [ terminal code ]

 {
   [ before (x) code ]
   [ after (x1,x2,... xn) code ]
   [ afterall (x1,x2,... xn) code ]

   [ newoption code ]
   [ at_option ([guard]) code ]
   [ at_state (label) code ]
   [ interrupt (guard) code ]

   [ translate x to expr ]
   [ divert ch1 to ch2 ]
   [ filter (ch!pattern) code ]
 }*
}
```

[...] denotes optional elements,
{ ...}* stands for 0 or more repetitions of a structure.

Figure 2: A roletype definition

code to or modify the code of the `proctype` it is applied to.

3.1.3 Components of a 'roletype' declaration.

The various clauses that may occur in a `roletype` declaration are listed with short explanations in Table 1. The following section explains their meaning in more detail.

The `uses` and `new` clauses should occur in this order at the beginning of the role-type declaration. Apart from that, the components of a role-type declaration may occur in any order; the integrator will apply the corresponding substitutions in the order given. E.g. the result of integrating

`after(x1)` *stmnt1*`;` `after(x1)` *stmnt2*`;`

will differ from

`after(x1)` *stmnt2*`;` `after(x1)` *stmnt1*`;`

in that in the first case *stmnt2* will end up before *stmnt1*, while the second case results in the reverse order.

code stands for a single PROMELA statement with a closing semicolon or for a sequence of statements enclosed in '`{`' and '`}`'. Only the code in the `initial` and `terminal` clauses may contain `run` statements.

expr is any expression evaluating to a value (in PROMELA all values are of integer type), i.e. anything that SPIN accepts on the right-hand side of an assignment. NB: The `before` and `after` clauses do not match *expr* to the right-hand side of the assignment but compare the resulting values by means of a conditional. E.g. applying `after(x=y+1) x==0;` will transform "`x = 5`" to "`x = 5; x = (x==y+1 -> 0:x)`."

***ch_1*, *ch_2*:** for the `divert` clause, ch_2 can be given by any expression that yields a channel. However, ch_1 may not be the result of an expression as, for example, in the code fragment

```
cc[(x==0 -> 1:0)]!x,
```

when ch_1 is given as `cc[0]`.

Remark: There is a way around this by extending such an array to contain the new channel ch_2. The statement

```
cc[((x==0 -> 1:0)==0 -> 2:(x==0 -> 1:0))]!x
```

would divert exactly the messages on `cc[0]` (= ch_1) to `cc[2]` (= ch_2). But then ch_2 may be given by a conditional expression, – or

`uses` *variables*	*variables* (local or global) are required in any `proctype` that the `roletype` is to be applied to
`new` *local variables*	*local variables* are introduced to the `proctype` that the `roletype` is applied to
`initial` *code*	*code* is added at the very beginning of the `proctype`
`terminal` *code*	*code* is added at the very end of the `proctype`
`before` `(`$x[= expr]$`)` *code*	*code* is added directly before assignments (of *expr*) to x.
`after` `(`$x[= expr]$`)` *code*	*code* is added directly after assignments (of *expr*) to x.
`afterall` `(`$x_1, \ldots, x_n$`)` *code*	*code* is added after all variables $x_1, \ldots, x_n$ have been assigned to
`newoption` *code*	*code* is added as a new branch in the main reactive loop
`at_option` `(`[*guard*]`)` *code*	*code* is added to every option guarded by *guard* in the main loop
`at_state` `(`*label*`)` *code*	*code* is added immediately after *label*.
`interrupt` *code*	*code* is added to the main reactive loop in an `unless` construct
`translate` *arg* `to` *expr*	in all send operations and assignments the *arg* is replaced by the expression *expr*;
`divert` ch_1 `to` ch_2	all send operations on ch_1 are changed to send operations on ch_2.
`filter` `(`*ch*`!`*pattern*`)` *code*	all matching send operations on channel *ch* are *replaced* by the given code; parts of the *pattern* can be used in the given code.

Table 1: The components of a `roletype` (2)

the expression yielding ch_2 would have to be evaluated in a separate statement, which breaks the atomicity of the send or receive operation. Another problem that this work-around will entail, is that the feature integration itself will introduce assignments to channel variables (and possibly channel "constants"), which will make it impossible to ensure consistency over several integration steps.

It would be most desirable to be able to write send (and receive) statements of the form

$$(\mathit{cond}\ \texttt{->}\ ch_1\ \texttt{:}\ ch_2)\texttt{!}msg,$$

but SPIN currently does not allow this.

guard can be any statement. In the `at_option` statement, the given guard statement can only be matched syntactically against those present in the code.

arg in the `translate` clause, stands for a variable or a symbolic constant (i.e. a value of type `mtype`), but not for a channel. (This is again due to the restriction which SPIN/PROMELA places on the syntax for channels.)

pattern in a `filter` clause consists of one or more fields, according to the message format defined for the channel *ch*. Each field can either be a constant (matching that *explicit* constant), the name of a variable in scope (matching that variable name), or the wild-card "`_`" which matches anything in that field, even complex expressions. Anything matched (except for "`_`") can be referred to in the *code* given with the `filter` clause using "`$`i" where i is the index of the field, counting from one; "`$0`" stands for the whole message. As in PROMELA, the two forms of send operations, "$ch\texttt{!}x, y, z$" and "$ch\texttt{!}x(y, z)$", are considered equivalent, so matching happens independently of the form chosen.

label is a standard PROMELA state label in the original program.

Some of the clauses warrant a more detailed description:

`new` ***variable declarations*:** Apart from the obvious introduction of completely new variables this is also used to redeclare existing variables with a larger scope, this is especially useful for extending arrays or declaring new constants of type `mtype`. NB: A `roletype`may redeclare *local* variables only.

`afterall` $(x_1,\ldots,x_n)$ ***code*:** This clause adds *code* after the values of all given variables have been computed. It regards choice constructs and certain kinds of loops as "atomic" computations (roughly similar to the notion of `atomic` sequences). This requires some understanding of our classification of loops, for which we refer the reader to the next section.

`at_option` **(*guard*)** ***code*:** As mentioned above, *guard* is textually compared to the guards in the main loop, and the code given replaces the branch(es) with guard given. The guard however remains, and the new code is added after it. In the *code* one can use the symbol "`$$`" for the original code of the branch (excluding the guard).

In the case where no *guard* is given, *code* is used for every option in the main reactive loop, but "`$$`" of course matches each option in turn, so constructs like:

```
:: old guard -> if
                :: (new guard) -> new code
                :: (!new guard) -> skip
                fi
                original code
```

or

```
:: old guard -> if
                :: (new guard) -> new code
                :: (!new guard) -> original code
                fi
```

can easily be realised.

`interrupt` ***code*:** *Code* is added to the main reactive loop in an `unless` construct, i.e at the end of the structure

```
do
:: original code ...
:: original code ...
od
```

the integrator adds

`unless{`*code*`}`.

When the integrator encounters an interrupt clause it automatically defines a label `continue` at the beginning of the main loop, so that it is possible to return from the interrupt code with the statement "`goto continue`."

`at_state` (***label***) ***code***: *code* is added immediately after *label*.

3.1.4 New processes.

A feature may introduce any number of *new* `proctype`s. These may be instantiated from the `init` process by giving the appropriate `run` statements in the `feature` section. These will be added at the *beginning* of the `init` process. On top of that, role-types may introduce `run` statements in their `initial` or `terminal` clauses.

Of course a feature may also create new instances of `proctype`s that are already the original system. This might, for example, be used to add fault-tolerance to a system, so that a result is only accepted by consensus (*cf.* the Byzantine agreement problem). The instantiation of such processes follows the same rules as for new `proctype`s.

3.2 Restrictions and other practical considerations

We assume that every process has the basic structure[1] detailed in Figure 3.

In principle, it is possible to integrate any feature into any system that provides `proctype`s with the right names and arities and that has variables matching those required by the feature and its role-types. However for many PROMELA programs, feature integration does not guarantee any consistency, not even in naming. Obviously, for every process definition we can write a feature that will render it useless; for example, adding an non-terminating loop to the front of the process.

We can however give some guidelines to the sort of programs that features can be added to with few side effects. These guidelines facilitate a static analysis of the control flow and of the structure of communications.

- Programs should not contain `goto` statements. (These make the detection of loops impossible.)

- `proctype`s should only have one reactive loop (at the top level), (alternatively: reactive loops *must not* contain `break` statements.)

[1] In [2] Katz, too, remarks that, in general you would have to transform processes to a certain form. He specifically points out the existence of a normal form for CSP processes.

```
proctype (parameters)
{
   [ variable declarations ]
   [ other initialisations ]
continue:
 [   do
     :: one choice
     :: another choice
      ⋮       ⋮
     :: yet another choice
     od
   [ unless { interrupt code [ goto continue ] } ]
 ]
   [ cleanup code ]
}
```

Figure 3: Structure of a process in the base system

- generic channel variables (type `chan`) should only be assigned once; channel variables of declared type (i.e. initialised variables) may never be overwritten. Processes should not communicate channels in any way, including via global variables. (This restriction forbids 'mobility'. Features and the feature construct still make sense when we allow mobility, but features that can be defined with the current features construct have no means to keep track of changing channel variables.)

3.2.1 Dealing with Loops

For the purpose of feature integration (and this prototype) we introduce a (somewhat artificial) distinction between two different kinds of loops: reactive loops and computational loops. (This distinction is problematic since it is rather fuzzy, see the paragraph "Borderline cases" below.)

Computational Loops. A *computational loop* computes one or more values and then terminates, i.e. control moves on to other code in the same `proctype`. Computational loops end after a (bounded) finite number of iterations, and they are often enclosed in `atomic` or `d_step` sequences because they represent 'internal' computations within a component, which take negligible time wrt. communication and synchronisation between different processes.

Ideally, computational loops would not involve communication or global variables; however, this restriction cannot be upheld in real life.[2]

Reactive Loops. A reactive loop never ends, i.e. it does not contain a `break` statement or a `goto` to a label outside the loop, and it cannot be preempted by an `unless` construct. Moreover, the loop's guards will usually involve operations or tests on channels or on global variables.

Borderline cases. A typical case of a 'reactive computational' loop (or 'computational reactive' loop) is the polling of a list of clients at regular intervals, e.g. to collect information about their status. Assuming a correct implementation, this loop should never be blocked (under normal circumstances) and it will terminate after a finite number of iterations to make its result available for further processing.

Another possibility is a 'computational' loop which can be interrupted by an external event (message or global variable). Even if that event does not occur, the loop will still end after a finite number of steps. This models the occurence of an interrupt or an exception. (PROMELA's `unless` construct provides an elegant way of coding something like this.)

A third case is that of a process which can take different roles: inside a never-ending outer loop there are two or more reactive loops which represent the different roles. The process can switch between roles by means of `goto`s or `break`s, if certain conditions arise (e.g. exception/interrupt). This would introduce a hierarchical structure into the model. For example, a phone might be modelled by an originating and a terminating call model within the same `proctype`: when the phone is idle, the process waits in the main loop until it either spontaneously chooses to originate a call, or there is an incoming call, in which case it enters the loop representing the terminating call model.

Treating computational loops. The idea of a 'computational loop' is that it encodes a *computation* rather than a *process*, and the result of this computation is assigned to some variable(s). Therefore a computational

[2] This is only partly due to limitations imposed by SPIN and the semantics of PROMELA. It is also a sign that the distinction between computational and reactive loops is rather artificial.

loop should be treated like a set of assignments to the variables that are assigned in the body of the loop.

This is where `after` and `afterall` differ significantly: whereas the application of an `after` inserts code immediately after each matching statement, the code given in an `afterall` clause is inserted only after the loop's closing `od`.

3.3 Conditionals and choices

For the application of `afterall` clauses the choice construct ("`if` ...`fi`") is treated similarly to (computational) loops: for `afterall`,

```
if
:: (x==0) -> y=1   (a)
:: else -> skip
fi  (b)
```

is treated like `y = (x==0 -> 1:y)`, so the new code is inserted at point (*b*), while an `after` clause would insert code at point (*a*). (The two code fragments above are indeed semantically equivalent if the choice construct is enclosed in an `atomic` or `d_step` sequence.[3])

3.4 Atomic and deterministic sequences

Sequences in the feature. The code fragments given in a role-type may contain `atomic` and `d_step` sequences, which will be inserted into the original code unchanged for `initial`, `terminal`, `cyclical` and `afterall` clauses. For `after` and `before` clauses, the sequence will be *extended* to include the matching assignment. For `d_step` we have to take a little more care, see below.

Sequences in the original code. If feature code is to be inserted immediately before or after an `atomic` or `d_step` sequence, i.e. the matching statement for a `before` or `after` clause is inside the sequence, the code will be added *inside* the sequence. Again we have to treat `d_step` sequences more carefully.

[3]This implies that our feature construct does not always respect semantic equivalence! However, there appears to be no way around this problem when working on the syntactic level; – something we already found in our work with the SMV system.

Deterministic sequences. These are subject to two exceptions to what was said in the last two paragraphs:

- Since send and receive operations inside a `d_step` sequence can lead to errors, the code from the `before` or `after` clause will be included in the sequence only up to a send or receive command.
- For `afterall` clauses, `d_step` sequences are treated as a single statement, i.e. the code given will always be added at the end of the sequence. (Again we view the `d_step` sequence as a single computation.)

When using `filter` and `divert` clauses, or if send and receive operations are part of a `d_step` sequence in the feature code, it is the programmer's responsibility to ensure the resulting code cannot lead to blocking statements inside `d_step` sequences.

3.5 Miscellaneous

Since `assert` statements serve the verification of the *unfeatured* system, they are removed when a feature is integrated.

`Printf` statements, which are not essential to verification, but very helpful in simulations, cause some problems: if a `printf` appears directly after a guard in the main loop, it has to be bundled with the guard in the matching for `at_option`, otherwise the new body for that branch might not work. We demonstrate this in the second example (4.2). Sadly, this solution will still not resolve all problems of this sort.

4 Examples

4.1 The lift system

We have modelled a simple lift system, with only one button on each floor. For each of the buttons inside the lift and on the landings there is a process which may simply press the button if it is not already pressed. The core of the model is, of course, the process which reads these requests and moves the lift up and down accordingly, opening and closing the lift doors as appropriate.

In contrast to Cassez [1] we do not model the lift cabin and controller separately, since our feature construct does not require a certain infrastructure. One could say our model is written with easy and quick implementation in PROMELA in mind, whereas Cassez's model tries to emulate the physical reality as closely as possible.

Features for the lift system We have implemented a few features for the lift system; here we describe the "Overload" feature, which will prevent the lift from moving if it is too full.

The `feature` section introduces a new flag `overloaded` to indicate if the lift is too full, it then instructs the integrator to use the role-type "`stopper`" to modify the `proctype` "`lift`". Finally it introduces a new `run` statement to the init process: the `proctype` "`in_out`" will be instantiated when we run the model.

The process "`in_out`" serves to simulate people entering or leaving the lift so that it is below or above its capacity. Obviously this should only happen when the doors are open. This new process also introduces an assertion to test if the lift really cannot move when it is overloaded.

The role-type "`stopper`" achieves this by forcing the doors to remain open whenever `overloaded` is true. (Obviously this relies on the fact that the original system does not allow the lift to move with open doors.) Since the code given is an `d_step` sequence, in the resulting code, any assignment to `doors` will be included in this sequence. Hence no other process will be able to detect that the feature – not the original controller – determines `doors`' value.

Other features for the lift, such as "Car Preference" (giving calls from inside the lift precedence over those from landings) and "Parking" (moving the lift to a certain floor when there are no requests pending) are slightly more complicated but not very difficult to code.

4.2 The telephone system

A telephone system differs considerably from the lift system in that it is essentially distributed and therefore relies on communication and synchronisation. We have taken a very simple model[4] of the "Plain Old Telephone System" (POTS). In this simplistic system of four telephones, all that a user (a phone) can do is establish a call to another phone – or get the "busy-tone" if that fails. Also, only the originator of a call may end the call.

Features for the telephone system The first "feature" one might like to add is that both partners in a call can choose to shut down the call. For lack of space we show just a short excerpt from the code for the POTS model (Fig. 5) and the parts of the feature "SymmEnd" (Figures 6 and 7) which achieves this end.

[4] Our model is based on code written by Dr. M. Thomas, University of Glasgow.

```
feature Overload
{
  new  bit overloaded;

  apply stopper() to lift();

  run in_out();
}

roletype stopper()
{
  uses bit doors;
  after (doors)
    d_step{
     if
     :: (overloaded) -> doors=OPEN
     :: (else) -> skip
     fi}
}

proctype in_out()
{
  uses byte position;
  new  short old_pos;

  do
  :: atomic{(doors==OPEN) ->
      if
      :: overloaded=1
      :: overloaded=0
      fi}
     /* NB: state can change here! */
     assert(overloaded -> position==old_pos : 1)
  od
}
```

Figure 4: Code for the “Overload” feature

```
:: (state==TCONNECTED) ->
     printf("Phone %d: tconnected(%d).\n", self, partner);
     atomic{
       hup[partner]?_ ->   /* wait for partner to hang up */
       event = on; dev = on; phon[self]?x; partner = null;
       state = IDLE
     }
```

Figure 5: Excerpt from the code for POTS

```
at_option (state==TCONNECTED)
     if
     :: $$  /* put original code for this branch here */
     :: atomic{
          hup[self]!1;  /* signal hang-up to partner */
          printf("Phone %d: tclose(%d).\n", self, partner);
          event = on; dev = on;
          phon[self]?x; assert (x == partner);
          partner = null; state = IDLE
        }
     fi
```

Figure 6: Excerpt from a feature for POTS

The base system (POTS) uses the one place channels `phon[]` (one for each phone) as semaphores and to indicate which phone is connected to which. The channels `hup[]` (again, one for each phone) are synchronous and serve to synchronise when one party (in the base system always the originator) hangs up, forcing the other party to eventually hang up, too. The local variables `self` and `partner` are indices to these arrays of channels, and otherwise are pretty self-explanatory; `x` serves as a dummy variable. Obviously, `state` marks the state of the phone call at important points in a call. Finally, `event` and `dev` are of no further interest here.

In this example we can see that `printf` statements need special attention: if the feature integrator grouped the `printf` with the body of the branch and put it in the place marked "`$$`", the process could easily get into a deadlock by choosing that branch of the `if` statement, when in fact the partner process might never issue a rendezvous offer on `hup[partner]`.

In Figure 7 we show the interrupt clause from the same feature: here we prepare the originating phone for the possibility that the partner may issue a 'hang-up' signal. We use a conditional handshake to ensure that the interrupt can only happen when the originating phone is in one of the states `OCONNECTED` and `OCLOSE`. To make sure that we use a *fresh* channel, we redeclare the array of channels `hup[]` in the feature declaration:

```
uses chan hup[];
new  chan hup[NPHONES+1];
```

where `NPHONES` was the original dimension of the array – and the number of phone processes in the model.

There is one minor complication in this code fragment: the interrupt code has to test the processes semaphore (`phon[self]`), since originator may have cleared it already, before the 'hang-up' signal occurred. One might find this problem by looking at the original code – or indeed, by using SPIN to debug the feature.

The code excerpts we have shown here constitute the core of the feature description, the only things missing are the syntactic framework and the declarations of the variables used.

Other features for POTS Other features for the telephone system have proved to be of varying degrees of difficulty; namely the "Call Forwarding" features were very easily implemented, whereas "Call Waiting" and "Call Forwarding" proved to be far more involved.

```
interrupt
   atomic{
    hup[((state==OCONNECTED || state==OCLOSE)
                           -> partner:NPHONES)]?_  ->
    /* terminating line has shut down call */
    printf("Phone %d: oclear(%d).\n", self, partner);
    event = on; dev = on;
    if
    :: (phon[self]?[eval(partner)]) -> phon[self]?x
    :: (!phon[self]?[eval(partner)]) -> skip
    fi;
    partner = null; state = IDLE;
    goto continue
   }
```

Figure 7: The `interrupt` clause from the "SymmEnd" feature

5 Conclusions

We have developed the feature construct to extend the idea of modularisation to *post hoc* extension of system specifications. By keeping the original description of the system separate from specifications of features, one can develop additional functionality more easily.

During the development process it is always possible to check whether the feature under development works with the base system, and with the base system plus other features. Our examples have also shown that the base system does not have to be written with features in mind for the feature construct to be applicable.

Having SPIN to check the success of feature integration proved very helpful, and we see this as a major advantage of our approach. We hope that the feature integrator will develop into a useful testbed for all sorts of systems that are readily modelled by concurrent processes. With a tool like this, specifiers and developers could test ideas and specifications in a plug-and-play fashion.

Some more specific points about the next steps with the feature construct. We are planning to undertake an extended case study, which, among other things, will allow us to compare the feature construct for PROMELA with that for SMV.

In this paper we have pointed out some problems which are in part due to particularities of PROMELA and SPIN, in part to nature of the problem. Mosst of these problems seem solvable, and we will try to overcome them without complicating the feature construct any more. In the same vein, there remains some work to be done to find out if the expressiveness of our "pattern matching" is adequate. (In the two systems we have investigated we had little use for the `filter` clause, since the examples did not involve any sophisticated protocol.)

We would also like to extend the expressiveness of the `apply` statement so that the specifier can apply different role-types to different instances of the same process definition. Again this involves a kind of matching; and this should be easy to understand as well as flexible and expressive.

Finally, on a more theoretical point, it would be very interesting to characterise a (non-trivial) subset of PROMELA for which the feature construct does indeed respect semantic equivalence (*cf.* footnote on page 13).

References

[1] Franck Cassez. A model to describe feature integration. submitted, April 1998.

[2] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.