Siegfried Löffler, Ecole Nationale Superieure des Telecommunications, Departement Reseaux, 46 Rue Barrault, 75634 Paris Cedex 13, France

Current address: Siegfried Löffler, Hewlett–Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ, UK

E-mail address: `loeffler@res.enst.fr`

Ahmed Serhrouchni, Ecole Nationale Superieure des Telecommunications, Departement Reseaux, 46 Rue Barrault, 75634 Paris Cedex 13, France

E-mail address: `ahmed@res.enst.fr`

It has to be mentioned that the compiler itself is not validated and most probably still contains quite a lot of bugs. Therefore the resulting implementations are not 100% validated.

Another thing which is left to do is a validation of the protocol that we use for the connection of clients and servers. A basic description of this protocol can be found in [**7**].

4. Conclusions

With PROMELA / SPIN, Holzmann has attacked two of the main factors inhibiting more widespread use of specification or validation tools, namely difficulty of use and the inherent limitations of the finite state reachability methods.

However, for the step to the final implementation, some difficulties remain. The major drawback of PROMELA in its current state is that the semantics are still not exactly specified, therefore allowing interpretations which may lead to problems when implementing the design. Another problem is the missing capability of the language to refine the specification in a way that suffices to describe details (like timeouts) of an implementation.

Our extended SPIN tool is usable for the rapid prototyping of validated implementations of communication protocols. Because of our different interpretation of the "`atomic`" statement, it is possible for the implementation to behave not exactly as expected. The generated implementation is not 100% validated because we had to add code for a scheduler and for external communications which has not been exhaustively verified.

The main application field we see for the implementations generated with the current version of our extended SPIN tool is the rapid prototyping of testing scenarios.

References

[1] Gerard J. Holzmann, AT&T, *Design and Validation of Computer Protocols*, Prentice Hall, 1991

[2] Gerard J. Holzmann, AT&T, *What's New in SPIN Version 2.0*, in "SPIN Documentation", 1995
`http://netlib.att.com/netlib/att/cs/home/holzmann-spin.html`

[3] Gerard J. Holzmann, AT&T, *V2. Updates*, in "SPIN Documentation", 1995
`http://netlib.att.com/netlib/att/cs/home/holzmann-spin.html`

[4] A. A. F. Loureiro, S. T. Chanson and S. T. Vuing, *FDT Tools for Protocol Development*, Forte '92, Lannion, France, 1992

[5] E. Najm, F. Olsen, Télécom Paris, *Reactive PROMELA*, 1st SPIN Workshop, Montreal, 1995

[6] Eric Moreau, Jérôme Paillet, *PROMELA Compilateur — Document Technique*, Rapport de Stage, Télécom Paris, 1994

[7] S. Löffler, *A Promela To C Compiler, Rapport de Stage*, Télécom Paris, 1996
`http://www.res.enst.fr/~loeffler/abstract.html`

[8] J. A. Chaves, *Formal Methods At AT&T - An Industrial Usage Report*, Forte 91, Sidney, 1991

[9] C. Jones, *Formal Methods And Their Role In Industry*, ASWEC 91, 1991

[10] R. P. Hautbois, P. de Saqui–Sannes, *Results And Viewpoints On The Use Of Formal Languages*, Workshop "Formal Methods, Modelling And Simulation For System Engineering", St–Quentin En Yvelines, France, 1995

[11] R. Groz, J. F. Monin, M. Phalippou, D. Vincent, *Current Application Of Formal Methods In France Télécom — CNET*, Workshop "Formal Methods, Modelling And Simulation For System Engineering", St–Quentin En Yvelines, France, 1995

External Communications

We already mentioned that we added external events to the PROMELA models. For the design of the implementations, this results in the necessity to find a way to specify such events in PROMELA. Since we did not want to change the PROMELA syntax by introducing new language constructs, we decided to implement such external events using channels. These "external" channels are specified exactly like normal PROMELA channels, the only difference is that we prefix their names with "`ext_`" in the specification. This allows us to use the existing tools for simulation and validation.

In order to simulate or validate a model which communicates with other systems via external channels, it suffices to include proctypes describing those external events into one single file used for simulation and validation.

The external channels also allow us to divide one PROMELA specification into implementations which run in multiple UNIX processes and communicate with each other via external channels. For this we have to create a PROMELA source code for each UNIX process to be generated. Using "`#include`" preprocessor directives all proctypes can then be included either in the source file which is used for simulation or in the source files which are used for the compilation of the implementation.

The connection of the UNIX processes is realized in a client–server architecture. The server always keeps track of the queue contents of the external channels in all connected UNIX processes. If a client wants to read from an external channel, it has to send a request to the server. On the other hand, the server notifies all clients if the contents of any of the external channels changes. To avoid having access problems with two clients reading from the same channel we limited read access to an external channel to one UNIX process per channel. If a UNIX process has read from a channel, thereafter no other UNIX process is granted read access to the same channel. This reduces the communication between the UNIX processes and makes them much faster. Nevertheless, multiple proctypes are allowed to write into the same channel queue.

For the inter–process communication between the UNIX processes, AF_UNIX domain sockets are used. This makes it easy to distribute the processes over multiple machines by changing the AF_UNIX domain into the AF_INET internet domain.

Upon compilation of a PROMELA model, one can specify whether the compiled UNIX process should be the server or a client by setting the appropriate switch. Since all channel queues are kept within the server, it is advisable to make the UNIX process the server which uses the external channels the most.

3. Future Work

The most important drawback in our extended SPIN environment is that it is not yet possible to use synchronous channels for the communication between UNIX processes. Almost as important is the missing capability for "sorted send" and "random receive" operations between UNIX processes, which were added to PROMELA with version 2.0 of SPIN. [**2**] [**3**]. Also introduced with SPIN Version 2.0 and not yet supported are some other language constructs like the "`d_step`" statement.

This example shows that although we are using the same abstraction, treat it with the same tool and even use the same internal data structures as in validation, it is still possible to interpret the semantics of PROMELA differently.

Priorities. In Version 2.5 of SPIN, Holzmann added a mechanism to define process priorities for use during random simulations [**3**]. Those priorities could be easily implemented in the scheduler, however they aren't yet. The priorities were implemented to improve the debugging facilities. For validation they are not used at all.

Timer Mechanisms

PROMELA was designed as a validation language. For the implementation of a protocol, there are some important requirements that are not yet covered by the language. For example it is absolutely necessary that one can define a timeout. Since the `timeout` statement in PROMELA has initially been designed to avoid the blocking of a proctype, in PROMELA it is not possible to specify any value for a timeout. For the validation it is sufficient to know that a timeout can occur in a certain state. If it can occur, it does not matter how long it may take until this happens since time is an element which does not exist in the validation.

The original `timeout` statement therefore is not supported for the implementation.

A solution to this problem could be to introduce a parameter that can be given to the `timeout` statement. This would have the advantage that existing PROMELA models could be modified very easily. On the other hand, it would have the disadvantage that a change to the language syntax would be necessary.

Since we did not want to change the syntax of PROMELA, we searched for another possibility. As a replacement, a timeout mechanism which uses message queues was implemented. This mechanism is close to the implementation of timers in SDL where timers basically are modeled as messages.

For the communication with this timeout mechanism that is implemented in the runtime scheduler, the following three message channel names, whose names are reserved, were defined:

1. "`set_timer`" This channel can be used to set a timer. The channel definition "`chan set_timer=[1] of { byte, byte };`" has to be used to define the channel in the model. Afterwards, a message consisting of a timer identifier and a value (in seconds) for the timer can be transmitted in a message to the implementation scheduler.
2. "`timer`" This queue is used by the implementation scheduler to send a message if the timer expires. The channel definition "`chan timer = [1] of { byte };`" has to be added to the PROMELA model. The message consists of the timer identifier that was sent through the `set_timer` channel.
3. "`del_timer`" This queue can be used to delete a timer before it expires. The definition for the channel is "`chan del_timer = [1] of { byte };`" The message should be the timer identifier that was used when setting the timer. In recursive proctypes, it is advisable to use the "`_pid`" Variable to compute a unique timer identifier.

To validate a model which uses those timer queues, an additional PROMELA proctype has to be added to the specification which describes the timer mechanism in the scheduler, i.e. reads and writes the timer queues.

all branches that it already tried to execute because an "`else`" statement that might be one of the branches is only executable if there are no other possible transitions. Since "`else`" is only executable if all other statements in the proctypes state aren't, the scheduler has to do the random branch selections until it has reached all other branches. If a completely random choice is used, some unexecutable branches are probably chosen more than once. Therefore, the scheduler uses more CPU time than necessary.

3. A third possibility is to choose the *first* branch to be executed in a random manner. Afterwards the remaining transitions can be tested sequentially. This costs almost no additional CPU time and has the advantage of introducing a random element into the implementation.

Since we wanted to have at least some random in our implementations, we decided to implement the third algorithm. This also allows us to use implementations for random simulations of the model.

Blocking Proctypes. If there are no possibilities to execute any of the branches in a certain state of a proctype, this proctype should block. If this is true for all proctypes in a UNIX process (or if the current sequence of transitions is "`atomic`") then it is not necessary to check continuously whether a transition has become executable. The only types of events that could change this state of the UNIX process are external messages from other processes or timeouts. So it is possible to block the UNIX process until an external message event occurs or a timer expires. This reduces the CPU load of the machine on which the implementation runs.

Atomic Sequences. In PROMELA, sequences of statements may be defined as "`atomic`" sequences. An atomic sequence should, from the point of view of the other proctypes, be seen as one single instruction that is not interruptible. During the execution of an atomic sequence, the scheduler must not switch to another proctype. However, starting with Version 2.0 of SPIN it is legitimate for an atomic sequence to block [**2**] [**3**]. In this case it should — since Version 2.0 — be allowed to switch to another proctype.

This is implemented differently in our scheduler. If an atomic sequence in the implementation blocks, the scheduler will stay in this proctype until it unblocks, even if this will never happen.

The reason why we decided to interpret the semantics of the "`atomic`" statement differently is that we wanted to add external events to implementations. However, if an implementation blocks in a certain state because it would have to wait for an external event, one can never know if this event will ever occur or not, because it is not specified in the same model. Therefore we can not know if it is necessary to leave the atomic sequence in order to avoid a deadlock.

A solution to this problem could be to prohibit the use of external channels inside atomic sequences. An easier solution is, in our opinion, to interpret the semantics as they were interpreted by older versions of SPIN, i.e. not allowing the change of proctypes inside atomic sequences at all. If a designer wants to allow a proctype change in a certain state, he should specify this explicitly by breaking the atomic sequence into multiple sequences which are separated by the operation in which the proctype change is to be allowed.

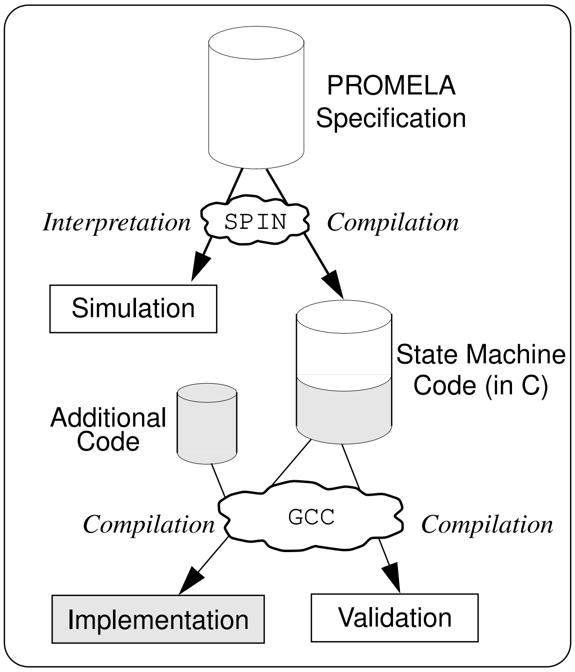

FIGURE 4. Code Dependencies With SPIN

Scheduling in the Implementation

The state machine plus the additional C code for the implementation are compiled into a UNIX program. Thus multiple proctypes which are designed to run in parallel are executed in one single UNIX process. For the switching between the proctypes a scheduler which selects the next active proctype and takes care of external communications and of timers is needed.

Non–Determinism. The most interesting feature of PROMELA as a language is the possibility to describe non–determinism. If non–deterministic choices are to be translated into executables (implementations), there are several possibilities for dealing with it.

In "`if..fi`" and "`do..od`" statements, all *executable* branches from the current state of the state machine can be chosen in an arbitrary manner. Before choosing a branch, the scheduler therefore has to test if it is executable. The following three strategies for the choice of the next transition can be imagined:

1. It must be allowed to simplify the implementation by just choosing *always the first executable* branch. The scheduler starts with the first transition, checks if it is executable, if it isn't, it tests the second one, and so on. This makes it very compact and usually quite performant. The behavior of the implementation will be the same each time it is started because there is no random element in scheduling (except possible external events).
2. Another possibility is to use a *random number generator* to choose between the branches. The disadvantage of this method is that the same transition branch may be chosen more than once. The scheduler has to keep track of

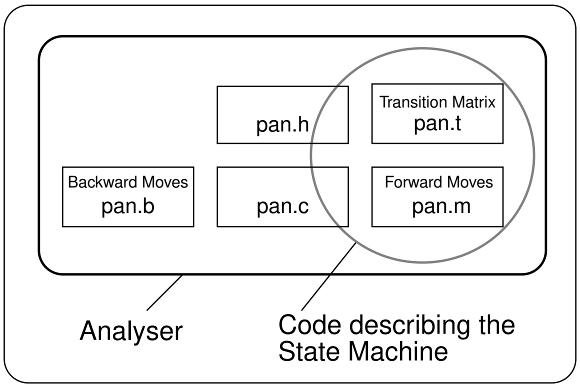

FIGURE 3. Overview Of The Code That SPIN Produces When Called With The Option "`-a`"

The two branches of the abstract tree melt together into one (as shown in figure 2) since the same abstraction is used for validation and implementation. This is the originality of our contribution that distinguishes it from most other tools for the generation of implementations, which are usually separate programs [**6**].

2. Our PROMELA to C Compiler

As we already mentioned, the analyzer which SPIN produces when called with the option "`-a`" is generated as C code. This code is structured in five files, as depicted in figure 3.

In order to re–use the state machine which SPIN generates corresponding to the PROMELA source we need the files "`pan.m`" (the forward moves) and "`pan.t`" (the transition matrix). In addition to those we use parts from "`pan.c`" (the main analyzer code) and "`pan.h`".

Figure 4 shows an overview over the data dependencies with our extended SPIN tool. The shaded regions in the figure represent the extensions we made for the generation of implementations. This is mainly the scheduler but also some code that allows us to define real–time timers and to communicate with other UNIX processes.

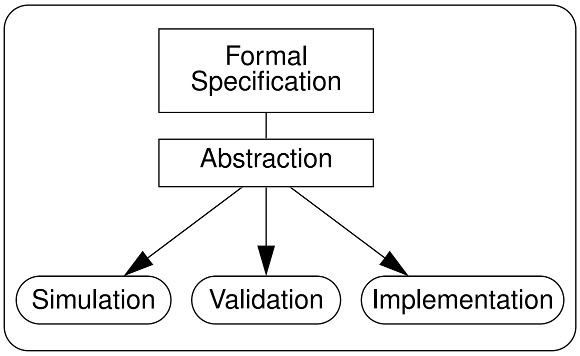

Figure 1. Standard Abstract Tree For FDT Tools

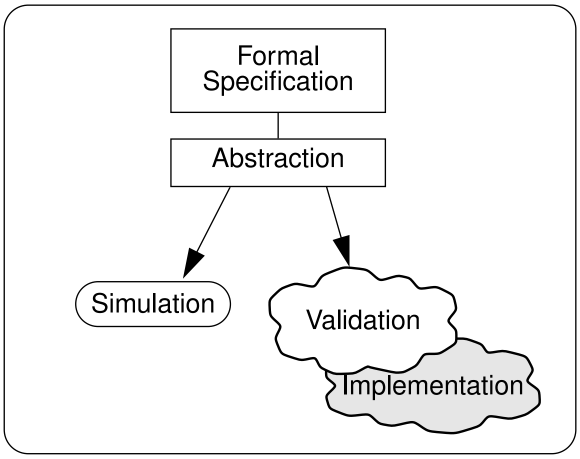

Figure 2. Using The Same Tool For Validation And Implementation

The main goal when creating an implementation from PROMELA source code obviously is to create a *validated* implementation which has certain provable characteristics. To achieve this it is necessary to keep as close as possible to the validated code, i.e. to stay as close to the validated abstraction in the tree in figure 1.

This was our starting point for the generation of the implementation. We went into the internal data structures of the validator generated by SPIN and tried to re–use as much of it as possible. The big advantage we had from using SPIN for this effort was that SPIN already generates C code for all transitions and actions. In fact Holzmann already mentioned this possibility to create implementations by proposing modifications to the validator as an exercise in his book [**1**] (p. 317, ex. 13–5).

By modifying the tool SPIN, isolating the state machine — the "motor" of the validator — and extending it with additional new code (mainly a scheduler), we achieve an implementation which has a very high fidelity to the validated code.

Creating Implementations from PROMELA Models

Siegfried Löffler and Ahmed Serhrouchni

ABSTRACT. SPIN is a tool to simulate and validate Protocols. PROMELA, its source language, is a formal description technique like SDL and Estelle that is based on communicating state machines. Unlike most other tools, SPIN is in the public domain and therefore is one of the most widely used formal verification tools today. PROMELA allows to specify distributed automata which can communicate using either message channels or shared memory. This contribution consists of an extension to SPIN which allows the creation of implementations from PROMELA specifications. This can be used for the creation of test scenarios and the rapid prototyping of validated protocol implementations.

1. Introduction

Formal Description Techniques (FDTs) are used in many fields of software engineering. One main application field is the validation of software designs, especially telecommunication protocols, but they are equally applicable to other fields such as avionics, nuclear power control, medicine, railway control etc. [**8**] [**9**] [**10**] [**11**].

Usually the FDTs are used for the validation of a design concept. This validation can be achieved either by simulations of the design or with special validation algorithms. After the concept has shown to fulfill the requirements, the formal specification usually is used as a guideline for the programmer for the creation of an implementation. However it can also be imagined that the formal specification is automatically translated into an implementation. This has the advantage of reducing the probability to introduce new design faults when translating it.

Such an automated generation of executable code from a formal specification is nothing new. Many efforts have been dedicated to this task in the past [**4**]. Usually different tools are used for simulation, validation and creation of the implementation. The tools take the formal specification as a starting point and use an abstract tree like the one shown in figure 1.

An advantage of PROMELA / SPIN [**1**] is that the same tool can be used for the simulation and the creation of the validator. Nevertheless, even with PROMELA / SPIN — although the same tool is used — the code that deals with the simulation is quite different from the code for the generation of the validator, since different data structures are created. For the validation, SPIN generates a validator which is generated as C source code, whilst for the simulation it directly interprets the PROMELA specification.