Protocol Design: From Specification to Implementation

Siegfried LÖFFLER, Ahmed SERHROUCHNI

Ecole Nationale Supérieure des Télécommunications, Paris, France
E-Mail: {loeffler, ahmed}@res.enst.fr

March 8, 1996

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 that can communicate using either message channels or shared memory. Because of the closeness to automata theory, its easy syntax and the intuitive graphical interface "xspin", PROMELA is also very suitable for educational purposes. On the other Hand, SPIN offers modern algorithms for protocol validation. This contribution consists of an extension of SPIN that allows to create a compiled implementation of the specification. This can be used for testing protocols and building test scenarios. As an example for the rapid prototyping of an implementation we specify the "Alternating Bit Protocol" in PROMELA, validate it with SPIN and afterwards compile it into a distributed implementation.

Keywords: PROMELA, SPIN, protocol design, formal validation

1.0 Introduction

In many fields of software development, there is a strong need for error free implementations. The formal description techniques (FDTs) provide means to designers of software that help them to ameliorate the quality of their products. Those techniques are widely used in the world of telecommunication systems, but are equally applicable to other fields such as avionics, nuclear power control, medicine, railway control etc. [9][10][12][14]. Many utilities have been developed to promote the application of FDTs in all fields of software engineering.

Usually, the software development process looks like shown in Figure 1. The problem is given to the developer as an informal, technical specification. He conceives how it could be solved, possibly using flowcharts or structural diagrams. Afterwards he translates his design into a pro-

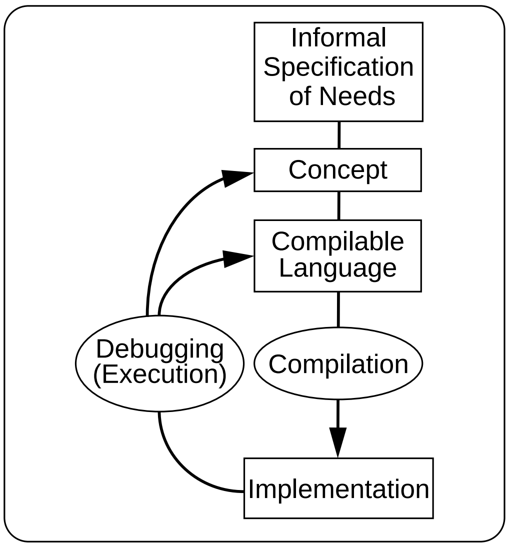

Figure 1: Usual Way of Software Development.

gramming language. If he discovers errors while executing his program, he goes back to the conception phase or the coding phase and corrects them. He loops through this cycle until he *thinks* that the software he produced is error free or at least fulfils the requirements. Many errors won't be discovered until the software is in real use.

Because of this, there is a strong need for a way to produce software with less errors. Many efforts in different directions have been made to improve the quality of software. One possibility is to use the formal description techniques (FDTs). They were originally designed for the validation of the conception of software, but can also be useful to validate implementations.

The common way of application of formal techniques in order to validate a formal specification is shown in Figure 2. After the informal specification has been written, a formal specification is created that can be validated and simulated with a tool. The formal specification is later used as a model for the implementation [7][8]. During the design of

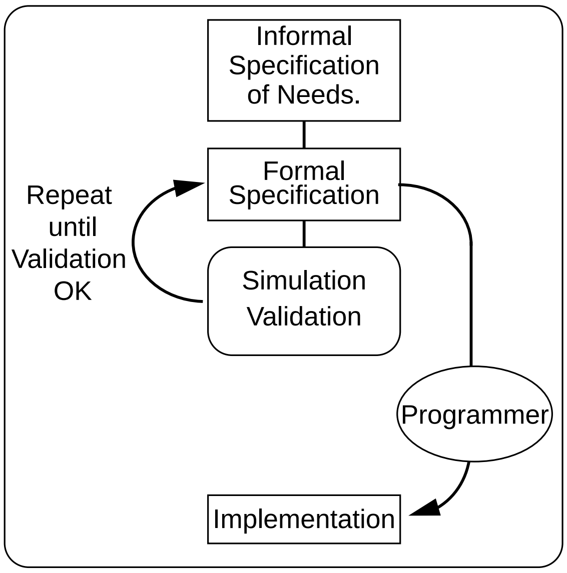

Figure 2: Software Development using Formal Description Techniques

the specification, it is possible to simulate the model in order to find errors. This is very useful for debugging the specification.

The same model is used for the simulation and for the validation. For the latter, special validation tools are used. After successfully being validated, the model is finally manually translated into an implementation by a programmer.

Unfortunately, the transition from formal specification to a compilable language holds many new possibilities to introduce errors [15]. First, the programmer is likely to make mistakes when translating it into a compilable language. Second, the compiler or the libraries that were used may have errors. So, by validating the formal specification, it is still not proved that the resulting implementation will fulfil the requirements.

In order to produce better code, the number of steps from specification to implementation has to be decreased. Especially, the steps in which errors are likely to be introduced should be avoided, automated or replaced by steps that can be validated.

The optimum would be to validate the implementation itself, i.e. to use the same code for validation and implementation. Figure 3 shows how the software development process can be organised to do this.

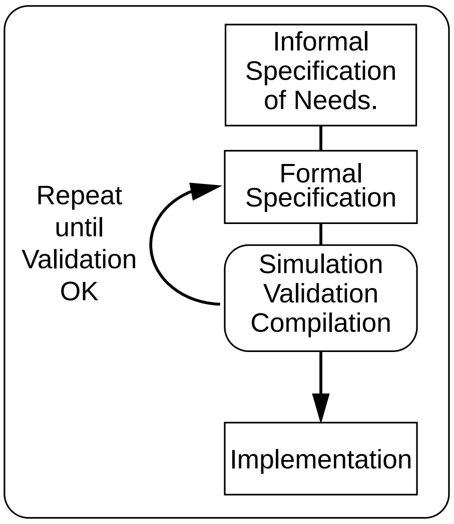
Informal Specification of Needs.
Formal Specification
Repeat until Validation OK
Simulation Validation Compilation
Implementation

Figure 3: Using the Validation Model to compile the Implementation.

Our contribution consists of an extension to SPIN that allows to use the "motor" of the validator that SPIN generated as a skeleton for the implementation.

2.0 PROMELA and SPIN

2.1 History of PROMELA / SPIN

PROMELA (Protocol Meta Language) was developed by Gerard J. Holzmann of AT&T. He also wrote the corresponding tool called "SPIN" (Simple Promela Interpreter, which he put in the public domain. PROMELA and SPIN are presented in Holzmann's book "Design and Validation of Computer Protocols" [1]. In his book, he does not only describe the types of coordination problems that a protocol designer has to deal with, but he also introduces PROMELA as a protocol design and specification technique. Included in the book is the source code for SPIN.

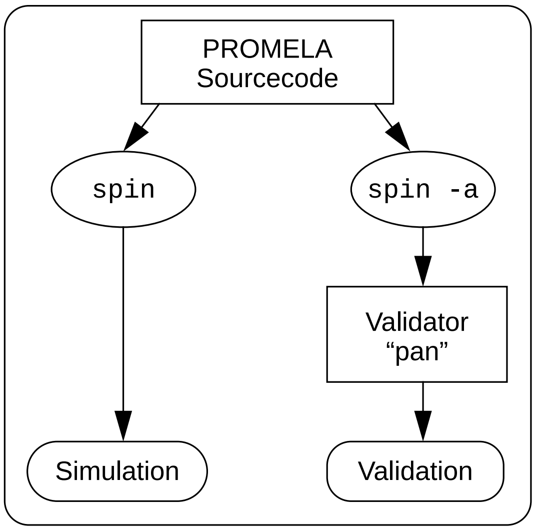

Figure 4: Structure of Holzmann's PROMELA/SPIN System

Figure 4 shows an overview of the system. The same tool, SPIN, is used for the simulation and the validation of the model. For the simulation, the PROMELA source is interpreted. For validation, it is compiled into a state space analyser. This analyser is coded in C and consists of a part describing a state machine and code that walks through a tree of all possible transitions in the state space. Multiple algorithms for validation, which are described in detail in the book, are implemented in SPIN.

2.2 The PROMELA Language

The PROMELA language is exceptionally easy to learn because it contains very few language elements. It is quite similar to conventional programming languages such as Pascal or C. However, PROMELA is mainly a "*protocol validation model*" language. At the validation level, the model does not have to describe the exact details of the implementation. The focus is on the *structure* of the model.

The most important design goal of PROMELA was the specification of distributed systems. Such systems are represented by sets of concurrent, parallel processes. The communication between those processes is performed by means of queues and/or shared variables.

The syntax of PROMELA is simple and compact but nevertheless surprisingly powerful in its expressive capability. The following three types of objects are used to construct a PROMELA specification:

1. **Processes**
   The process uses an extended finite state machine model for the description of its behavior. The concepts of the processes are very close to CSP [11][13].
   All processes are on the same level. Processes are described by the `proctype` construct, which introduces a process prototype. A special "`init`" process is always invoked by the system at start-up. Additional processes can be dynamically invoked during execution with the "`run`" statement. The same process prototype can be invoked multiple times, this makes it possible to build recursive models.

2. **Channels**
   The channels are an exceptional data type. They are essentially finite-length FIFO queues. Channels can be defined globally for the specification or locally within each proctype. A channel definition contains the data types of the messages that can be transmitted over a channel. The data types may contain channel names, what can be very useful for recursive models.
   Channels may either be defined as synchronous ("rendez-vous") or as asynchronous. Synchronous channels are represented by defining the length of the channels FIFO queue as zero and can be used to synchronise two processes.
   Non-FIFO random access to the queues is possible using the primitives for "sorted send" and "sorted receive". They allow to insert/grab data into/from any place in the queue.

3. **Variables**
   Like channels, variables can be defined either globally or locally. By defining them globally, they can be used to interchange data between the different proctypes. This can be used to describe a communication via shared memory.

```
proctype ProcessA(int x, bool flag)
{
  /* Body of ProcessA's definition */
}

proctype ProcessB(byte y)
{
  /* Body of ProcessB's definition */
}

init
{
    run(ProcessA(5,true);
    run(ProcessB(3));
    printf("Init done.\n")
}
```

Figure 5: Typical Skeleton of a PROMELA Program.

Figure 5 shows a skeleton of a typical PROMELA program. This program consists of two (empty) process prototype definitions. The `init` process is invoked when the program is started and instantiates the two processes by executing the run statements. Afterwards it writes a message on standard output.

2.3 Control Structures

For the validation of a specification, the most important aspect is that nothing must be considered as impossible. If it is possible for an event to occur, it always has to be considered that it actually may occur, even if the probability is very small.

Figure 6 shows a state machine with a non-deterministic choice. If, for example, the value of the variable `a` is `1`, all

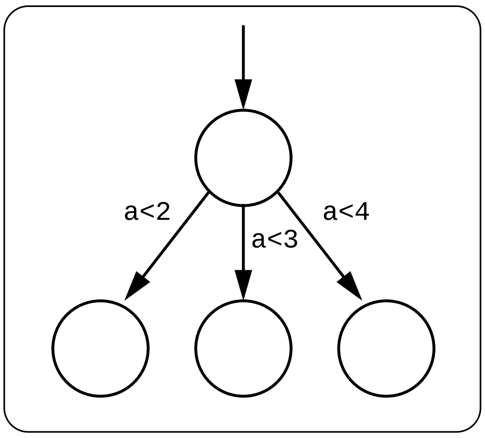

Figure 6: Non-Deterministic Possibilities in a Finite State Machine

three expressions are valid and therefore all three transitions are executable. The corresponding PROMELA control structure is shown in Figure 7. During the simulation,

```
if
:: (a<2) -> ...
:: (a<3) -> ...
:: (a<4) -> ...
fi
```

Figure 7: Non-Deterministic Possibilities in PROMELA

SPIN will choose between the different executable statements in each “`do..od`” or “`if..fi`“ statement in a non-deterministic manner. In validation, a tree of all possible transitions in the state space is walked through.

2.4 The Tool SPIN

The ultimate goal of SPIN and indeed of all testing or validation methodology is to demonstrate, with some degree of confidence, that a proposed design or implementation meets its requirements. Traditional methods for doing this include computer simulation studies, physically testing an implementation for conformance to its specifications, design review committees, beta testing etc. A computer-based methodology attempts to make this process much faster and more economical. Some criteria for evaluating a computerised validation tool might therefore include the following:

1. What questions can be posed, i.e. what criteria can be verified?
2. What degree of confidence will the system provide?
3. What is the cost, in terms of human and physical resources, of using the tool?

The *possible verification criteria* when using SPIN are:

- Is the system free of deadlocks?
- Can any of the assertions, which may be inserted by the user at different points in the program, ever be violated during execution?
- Can the system halt in unauthorised states?
- Are there any useless ‘non-progress cycles’, i.e. livelocks?
- Are there portions of the source code which are never executed?
- Does the system statisfy a user-supplied temporal logic formula?

The *degree of confidence* one has in a system might depend on the number of users the system has. The more users there are, the more probable it is that most bugs already have been found. SPIN today has about 2000 users worldwide, and the number is increasing daily. SPIN allows to choose between two different validation algorithms. With the exhaustive full state space analysis a 100% coverage of all possible transitions can be achieved. Using the supertrace algorithm a bit state technique is applied. This verification algorithm does not cover the full state space but it uses much less memory and therefore makes it possible to validate much larger models. SPIN gives information on the percentage of the state space that has been verified. With this information, the degree of confidence in the validation can be estimated.

The *cost in physical resources* is relatively low because the PROMELA/SPIN system is free. The memory usage for the validation is low if one uses the modern supertrace algorithm. CPU time is not critical, usually the validation process takes less than ten minutes for models that use most of the available memory of 64 MB machines.

The language is easy to learn, so the *cost for human resources* that are needed in order to learn it is not too high. On the other hand, since the language constructs are pretty simple, it may take a longer time to design a specification than it does with other, more sophisticated FDTs.

Since the language is very close to finite state machines it is relatively easy for a programmer to translate the PROMELA specification into an implementation. With other, more abstract FDTs this translation can be much more difficult.

3.0 Creating the Implementation

The generation of executable code from a formal specification is nothing new. Many efforts have been dedicated to this task in the past [6]. Most of these efforts take the formal specification as a starting point and use an abstract tree like the one shown in Figure 8.

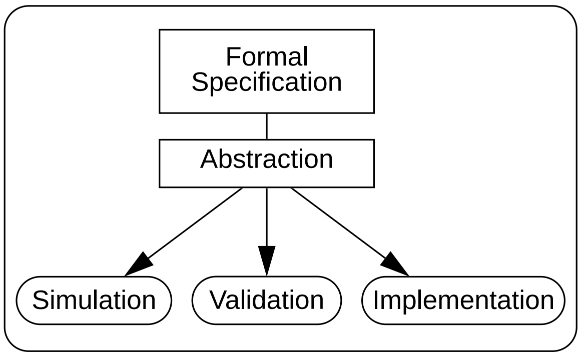

Figure 8: Standard Abstract Tree for FDT Tools

Usually different tools are used for simulation, validation and creation of the implementation. In Holzmann's PROMELA/SPIN the same tool is 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.

In order to keep as close as possible to the validated code, we did not only use the same tool for the creation of the validator and the implementation but we went into the internal data structures of the validator in order to reuse parts of it. We took SPIN and isolated the state machine and transition table - the "motor" of the validator - and extended it with additional code. Doing so the implementation has a high fidelity to the validated code. The two branches of the abstract tree melt together into one, as shown in Figure 8. An overview over the data dependencies in the resulting extended PROMELA/SPIN environment is given in Figure 10.

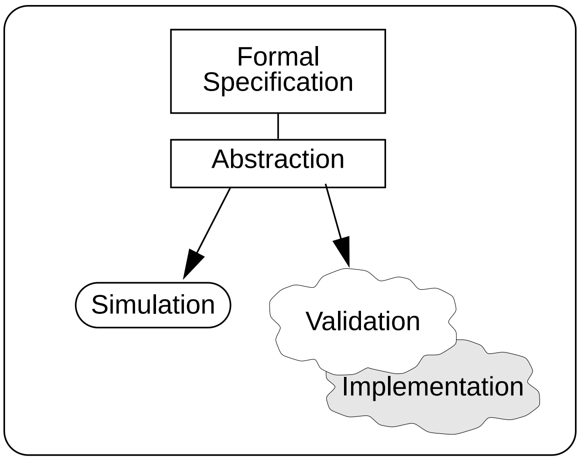

Figure 9: Using the same Tool for Validation and Implementation

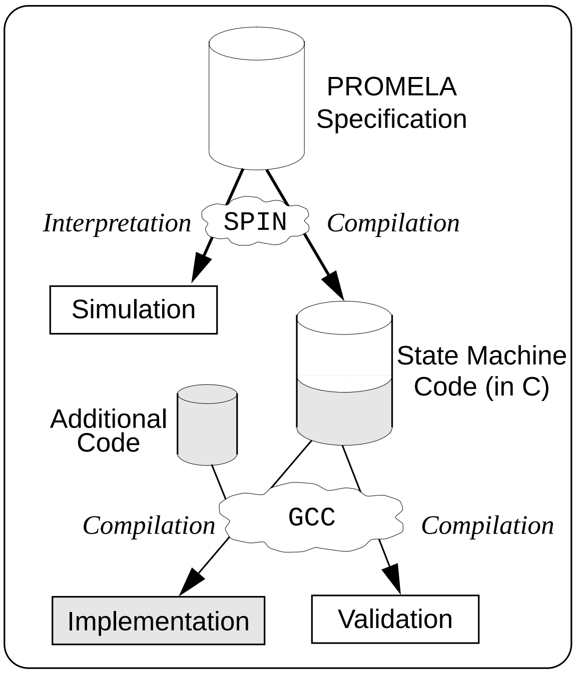

Figure 10: Code Dependencies with SPIN

The analyser that SPIN produces when called with the option "`-a`" is generated as C code and consists of the following parts:

1. A File called "`pan.m`" containing the forward moves for the state machine of the translated PROMELA specification.
2. A File called "`pan.t`" in which the transition matrix is defined. The transition matrix contains the information about which transitions are possible from one specific state to other states.

3. A File “`pan.b`” that defines backward moves. Backward moves are used during validation to ascend the tree of the state space.
4. Finally, the validator Code itself is contained in the Files “`pan.c`” and “`pan.h`”.

Figure 11 shows the files that SPIN produces when it is called with the option “-a” in order to produce the analyser. Here SPIN already generated the state machine corresponding to the PROMELA source. This is our starting point for the generation of the implementation. In order to create an implementation that uses this state machine, some kind of scheduler that chooses the executable transitions and executes the corresponding moves is needed in addition to “pan.m” (the forward moves) and “pan.t” (the transition matrix). For the development of protocols possiblities for external communications and real-time timers would be desirable.

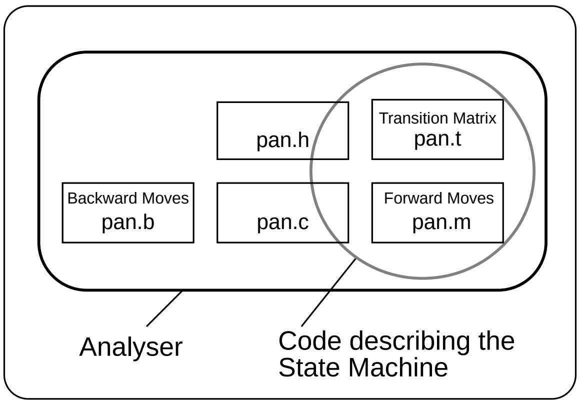

Figure 11: Overview of the Code that SPIN produces when called with the Option “-a”.

In the following section we describe the extensions that we have made to the state machine in detail.

3.1 Extensions to the State Machine

The state machine plus the additional C code for the implementation are compiled into a UNIX program. Thus, multiple proctypes that are designed to run in parallel will be executed in one single UNIX process. For the switching between the proctypes, a scheduler is needed that selects the next active proctype and takes care of external communications and of timers. Figure 12 shows the structure of the UNIX process that is produced as implementation of the model in Figure 5.

3.1.1 Scheduling

The most important element that distinguishes PROMELA from other languages is the *non-determinism*.

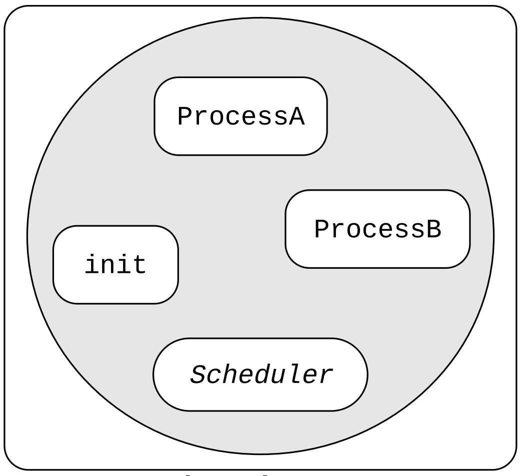

Figure 12: The resulting UNIX Process

In Figure 6 and Figure 7 we presented an example for non-determinism on the specification level. The interesting question now is how this can be translated into an implementation.

When executing the program, there are several possibilities to deal with the non-determinism. Since the validation was made for all possible choices in non-deterministic statements, any executable choice can be selected without the risk of introducing new errors.

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 goes to 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 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 all branches that it already tried to execute because an “`else`” statement[1] 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.

1. The “`else`” statement was added in SPIN Version 2.0, see [4]

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

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, 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 reception or send has been executed or a timer expires. This reduces the CPU load of the machine on which the implementation runs.

3.1.1.1 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. Starting with Version 2.0 of SPIN, it is legitimate for an atomic sequence to block[1]. In this case it should - since Version 2.0 - be allowed to switch to another proctype. This is implemented differently in the 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.

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.

3.1.1.2 Priorities

In Version 2.5 of SPIN, Holzmann added a mechanism to define process priorities for use during random simulations[2]. 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.

1. See section 2.3.1 in [4]
2. See [5] for details on priorities.

3.1.2 Timer Mechanisms

PROMELA was designed to be 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 after what time this can happen as time is an element that 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 has the advantage that existing PROMELA models can be modified very easily. On the other hand, it has the disadvantage that a change to the language itself is necessary.

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

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

1. “`set_timer`”
   This channel can be used to set a timer. The 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 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[1] to compute a unique timer identifier.

To validate a model that uses those timer queues, an additional PROMELA proctype has to be added to the specification that describes the timer mechanism in the scheduler, i.e. reads and writes the timer queues. An example how this can be done is shown in Figure 22.

3.1.3 External Communications

The implementation of a protocol does not only consist of the specification of the protocol itself but also has to provide a possibility to communicate with other entities like depicted in Figure 13. For validations of other specifications than protocols it might also be interesting to have the possibility to spread the implementation over multiple UNIX processes or even over multiple machines in a network.

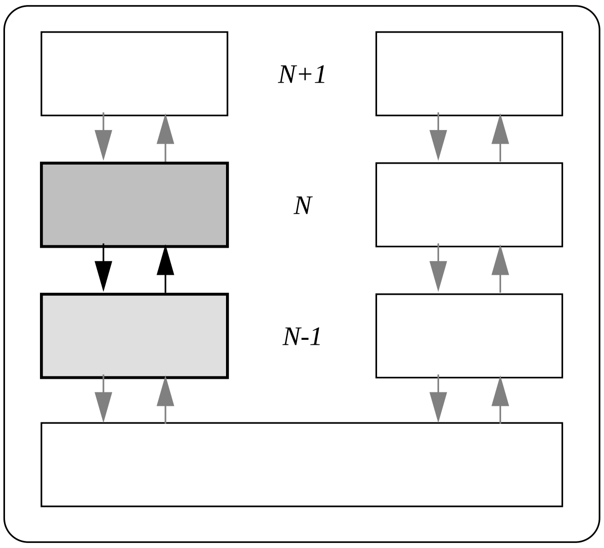

Figure 13: Two communicating Protocol Entities

Since we did not want to modify the syntax of PROMELA we had to find a way to mark communication channels as external channels. We decided to do this by prefixing the channel name with "`ext_`". Doing so, the specification can still be simulated and validated with SPIN. It suffices to include the proctypes of all UNIX processes that are to be generated into one single file that is used for validation like depicted in Figure 19.

1. This variable was introduced with Version 2.0 of SPIN. It can be used to identify the proctype that is currently executed. (See section 2.2.2 in [4]).

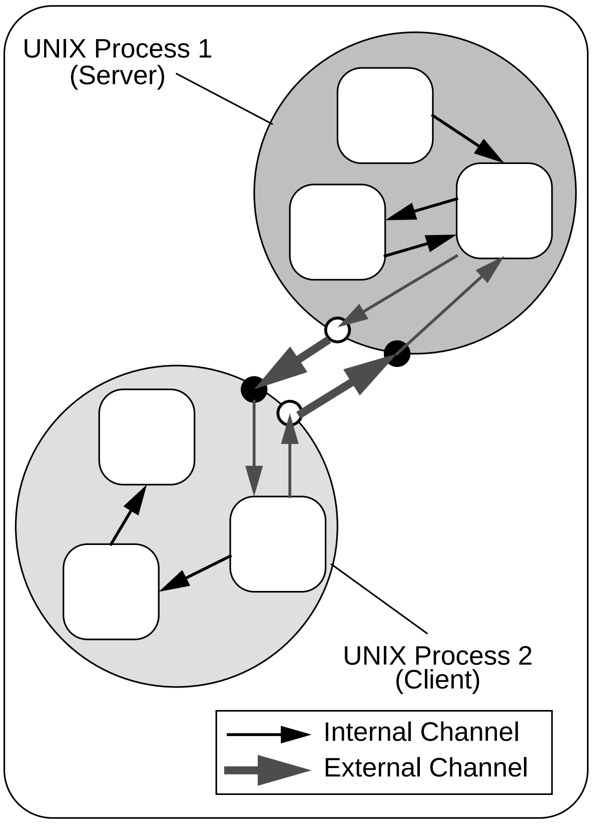

Figure 14: How external Channels can be used to connect UNIX Processes

For the connection of the UNIX processes, we've chosen a client-server architecture since we did not want to limit the number of connected processes. The server always keeps track of the queue contents of the external channels in all connected UNIX processes. If any of the clients 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, the read access to an external channel is limited to one UNIX process, i.e. if one 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 processes and makes it 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. Therefore it is easy to spread 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. If no external communications are needed the communication code is not included into the implementation.

Currently, there is no support for communication between the UNIX processes using shared memory.

3.2 Incompatibilities

Some new features of SPIN Version 2.0 are not yet implemented for the implementations: The "`active`" and the "`D_step`" statement are not yet supported.

3.2.1 Synchronous (Rendez-vous) Channels

External synchronous channels are not yet implemented. Internal synchronous channels work as in simulation and validation, i.e. proctypes performing a read or write operation on a synchronous channel are blocked until the corresponding other operation is carried out on the same channel.

3.2.2 Claims

Claims such as `never()` or `assert()` are only useful for the validation of the model. Therefore they are not supported in the implementation and must be removed from the model before creating the implementation. This can easily be done by using preprocessor directives.

3.3 C Interface

Since the compiled PROMELA models use UNIX sockets for communication, it is not too hard to write programs in C that communicate with the implementation. This is an important aspect because the compiled implementation is often needed to build validated prototypes of protocol entities that are used to test an implementation of a protocol on the other side of the communication channel. The easiest way to build such a test scenario is to create a validated communication server by compiling the PROMELA model. Afterwards the implementation that is to be tested can be connected as a client. For the communication with the compiled server, the same routines as in a compiled PROMELA client can be used. These offer possibilities to peek into the contents, read from and write to a channel. All channels are identified by the name they have in the PROMELA model, so each access to a channel includes the channel name (as a character string) as an argument. Since the server sends out notification messages after each change in the data structures, those messages have to be received by all clients. Therefore, it is mandatory for all clients to deal with those messages, i.e. the procedures for updating the data structures must be called in regular intervals. In a future release we plan to furnish procedures that convert the messages from the internal format into a user definable format.

4.0 An Example

As an example for a compiled distributed implementation, we used Lynchs "Alternating Bit Protocol" (ABP) [16]. The sender and the receiver were modeled in two PROMELA proctypes shown in Figure 15 and Figure 16. They are kept in separate files to allow the creation of a distributed environment. For simulation and validation they are included, together with the necessary definitions of the global resources (channels) into one single file (as shown in Figure 19). For the creation of two separate UNIX processes, two other files that include the channel definitions, the proctypes and an initial proctype are used.

```
proctype Sender()
{
byte any;
do
:: do
   :: set_timer!_pid(3); ext_receiver!msg1;
      printf("MSG1-->\n");
      if
      :: timer??_pid; printf("timeout1\n")
      :: ext_sender?ack1; del_timer!_pid;
         printf("ack1<-\n"); break
      :: ext_sender?any; del_timer!_pid;
         printf("(lost)<-\n")
      fi
   od;

   do
   :: set_timer!_pid(3); ext_receiver!msg0;
      printf("MSG0-->\n");
      if
      :: timer??_pid; printf("timeout0\n")
      :: ext_sender?ack0; del_timer!_pid;
         printf("ack0<-\n"); break
      :: ext_sender?any; del_timer!_pid;
         printf("(lost)<-\n")
      fi
   od;
od;
}
```

Figure 15: The PROMELA Model for the Sender

```
chan set_timer = [1] of { byte, byte };
chan del_timer = [1] of { byte };
chan timer = [10] of { byte };
```

Figure 18: Includefile "timers.h" for Timer Channel Definitions

```
proctype Receiver()
{
byte any;
do
::do
  :: ext_receiver?msg1; printf(“\t\t->msg1\n”);
     ext_sender!ack1; printf(“\t\t<-ACK1\n”);
     break
  :: ext_receiver?msg0; printf(“\t\t->msg0\n”);
     ext_sender!ack0; printf(“\t\t<-ACK0\n”)
  :: ext_receiver?any; printf(“\t\t(lost)\n”);
  od;

  do
  :: ext_receiver?msg0; printf(“\t\t->msg0\n”);
     ext_sender!ack0; printf(“\t\t<-ACK0\n”);
     break
  :: ext_receiver?msg1; printf(“\t\t->msg1\n”);
     ext_sender!ack1; printf(“\t\t<-ACK1\n”)
  :: ext_receiver?any; printf(“\t\t(lost)\n”)
  od;
od;
}
```

Figure 16: The PROMELA Model for the Receiver

```
mtype = {msg0,msg1,ack0,ack1};
chan ext_sender =[2] of { byte };
chan ext_receiver=[2] of { byte };
```

Figure 17: Includefile “mychans.h” with Definitions of all global Channels

```
#include “timers.h”
#include “simtimers.h”
#include “mychans.h”
#include “send.pr”
#include “recv.pr”
init
{
  atomic{ run sender(); run receiver();
          run simtimers()}
}
```

Figure 19: PROMELA Sourcefile for Simulation and Validation

```
#include “timers.h”
#include “mychans.h”
#include “send.pr”

init
{
  run sender()
}
```

Figure 20: PROMELA Sourcefile for the Implementation of the Sender

Those two files are shown in Figure 20 (for the sender) and Figure 21 (for the receiver). Both of the compiled processes may either be the server process or the client process. This may be chosen by the user by setting the appropriate switch during compilation.

```
#include “timers.h”
#include “mychans.h”
#include “recv.pr”
init
{
  run receiver()
}
```

Figure 21: PROMELA Sourcefile for the Implementation of the Receiver

```
proctype simtimers()
{
byte a,b;
set_timer?a,b; run simtimers();
if
 :: del_timer?a;
 :: timeout; timer!a;
fi
}
```

Figure 22: PROMELA Source “simtimers.h” that describes the new Timers for Simulation / Validation

5.0 Conclusions

In producing PROMELA/SPIN, Holzmann has attacked two of the main factors inhibiting more wide-spread use of specification or validation tools, namely, difficulty of use and the inherent limitations of the finite state reachability methods. Some difficulties however remain. The major drawback of PROMELA in its current state is that the semantics are still not exactly specified, therefore allowing interpretations that may lead to problems when finally implementing the design. Another difficulty that is not solved yet is the missing capability of the language to refine the specification in a way that suffices to describe the details of an implementation. Nevertheless, our contribution can be used for the rapid protoyping of validated implementations of communication protocols or other PROMELA specifications. Because of the missing exact definitions of the PROMELA semantics it is still possible for the implementation to behave not exactly as expected. The created C code for the prototypes is not as readable as a manually created implementation, but the prototypes can be extended with own code for the external communications. This allows the creation of scenarios for the testing of other implementations.

6.0 References

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

[2] Omar Rafiq, *Histoire et problématique des réseaux informatiques*, in “Réseaux de communication et conception de protocoles”, Hermès, Paris, 1995

[3] Eric Moreau, Jérôme Paillet, *PROMELA Compilateur - document technique*, Rapport de Stage, Télécom Paris, 1994

[4] Gerard J. Holzmann, AT&T, *What’s New in SPIN Version 2.0*, In “SPIN Documentation” on http://netlib.att.com/netlib/att/cs/home/holzmann-spin.html, 1995

[5] Gerard J. Holzmann, AT&T, *V2.Updates*, In “SPIN Documentation” on http://netlib.att.com/netlib/att/cs/home/holzmann-spin.html, 1995

[6] A. A. F. Loureiro, S.T. Chanson and S.T. Vuing, *FDT Tools for Protocol Development*, In Forte ‘92, Lannion, France, 1992

[7] C. A. Vissers, R. L. Tenney and G. v. Bochmann, *Formal Description Techniques*, Proceedings of the IEEE, vol. 71, no. 12, pp. 1356-1362, 1983

[8] G. v. Bochmann, *Usage of Protocol Development Tools: The Results of a Survey*, in “Protocol Specification, Testing and Verification VII, H. Rudin and C. West (editors)”, North-Holland, 1987

[9] J. A. Chaves, *Formal Methods at AT&T - An Industrial Usage Report*, in “Forte ‘91”, Sidney, 1991

[10] C. Jones, *Formal Methods and their Role in Industry*, ASWEC ‘91, 1991

[11] C. A. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

[12] R. P. Hautbois, P. de Saqui-Sannes, *Results and Viewpoints on the use of Formal Languages*, in Workshop Formal Methods, Modelling and Simulation for System Engineering, St-Quentin en Yvelines, France, 1995

[13] B. Johnston, A. Serhrouchni, *PROMELA/SPIN: A Specification Language and a Validation Tool for Communication Protocols*, in Workshop Formal Methods, Modelling and Simulation For System Engineering, St-Quentin en Yvelines, France, 1995

[14] R. Groz, J. F. Monin, M. Phalippou, D. Vincent, *Current Application of Formal Methods in France Telecom - CNET*, in Workshop Formal Methods, Modelling and Simulation for System Engineering, St-Quentin en Yvelines, France, 1995

[15] G. v. Bochmann, *Protocol Specification for OSI*, in Computer Networks and ISDN Systems 18, North Holland, 1990

[16] W. C. Lynch, *Reliable Full Duplex File Transmission over Half Duplex Telephone Lines*, in Comm. of the ACM, Vol. II, No. 6, pp 407-410, 1968