TELECOM PARIS

UNIVERSITÄT STUTTGART

Deuxième Projet d'Etudes
Zweite Semesterarbeit

From Specification to Implementation: A PROMELA to C Compiler

Siegfried LÖFFLER

Ecole Nationale Supérieure des Télécommunications
Département Réseaux
46, rue Barrault
75013 Paris
France

E-Mail: loeffler@res.enst.fr

1 Introduction

Today, more and more things are controlled using microprocessors. About twenty years ago most people didn’t even have an idea of what a processor is, today almost everyone uses microprocessors - often without even noticing it. Microprocessors are used in cars, planes, trains, even in dishwashers, telephones, cameras,... - the list is endless. More and more they are used in fields where security is a very important aspect. A modern car for example already has many microcontrollers for various tasks, and the next car generation will even use communication networks to connect those controllers. It is obvious that failures of the controllers can be very dangerous in such applications.

Therefore there is a strong need for methods to ameliorate the quality of such systems. The term “quality” herein affects not only the hardware. There is also a very high potential for failures in the design or the realisation of the sytems software. Many efforts in different directions have been made to ameliorate the development process of such complex systems. Examples are technics that help structuring the design, object oriented programming languages, formal description techniques etc.

Our work focuses on verified software implementations and especially on implementations of communication protocols. We extend the PROMELA/SPIN system (one formal description technique) in a way that makes it possible to create compiled implementations of provable correct specifications.

1.1 Formal Description Techniques

The formal description techniques (FDT's) are a technique that can be used by designers of software to ameliorate the quality of their products. Those techniques were initially developed for and are mainly used in the world of telecommunications, but they are getting more and more important for other fields of engineering, like avionics, nuclear power control, medicine, railway control, automotives etc [9][10][12][14].

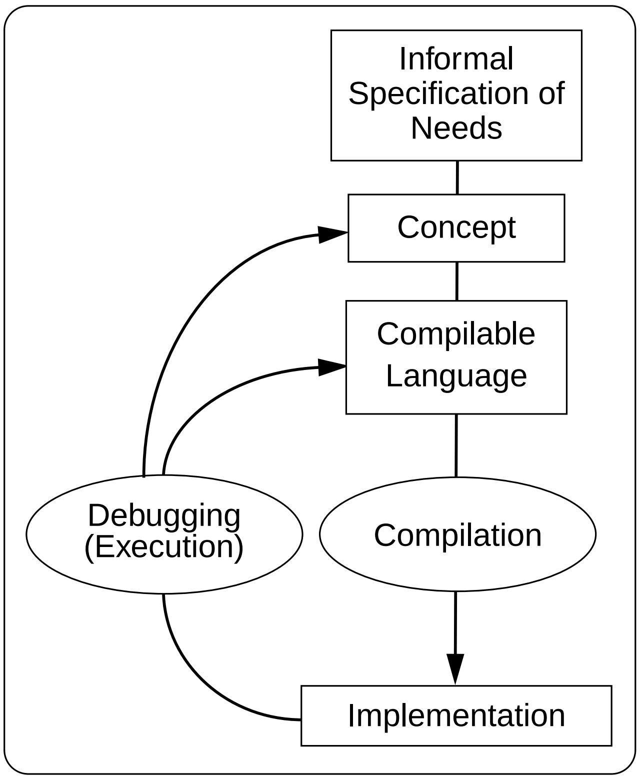


Figure 1: Usual Way of Software Development.

Usually, without using formal methods, the software development process looks like shown in figure 1. The problem is given to the developer as an informal specification. This specification very often is written as a technical document which may very well be interpreted and often does not cover important questions that are discovered during the development process. The task of the developer is to find a solution to this specification and to develop a structure for this solution. He might possibly use flowcharts or structural diagrams to organise his work. Afterwards he translates his design into a programming 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.

The FDT's try to tackle this problem by assisting the developer already during the design phase. The common way of application of formal techniques in order to validate a formal specification is shown in figure 2. The developer is again given an informal specification. Now, once he is aware of the out-

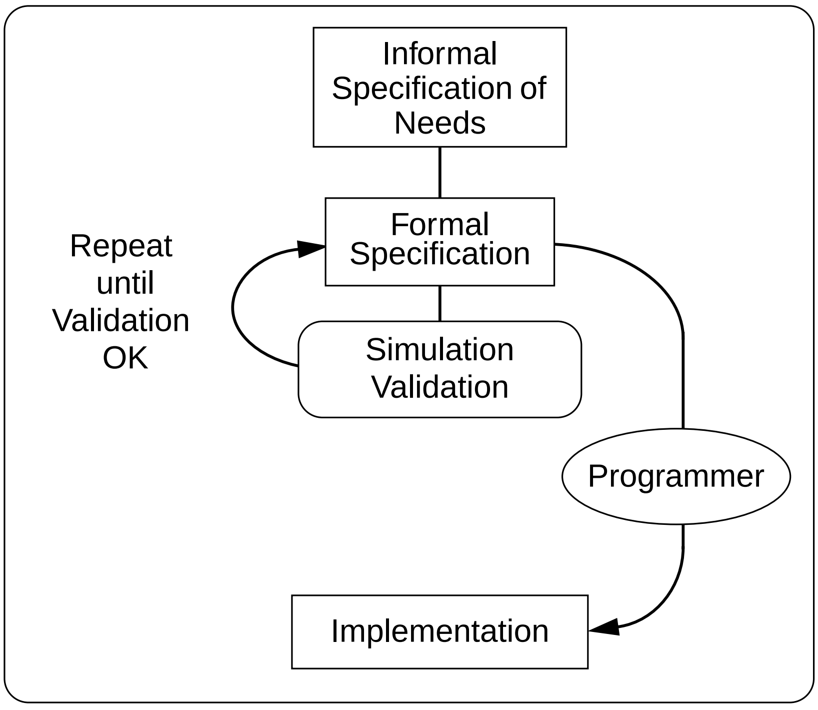


Figure 2: Software Development using Formal Description Techniques

line of his conception, he formulates the skeleton of his specification in the formal language. This formal specification can be validated and simulated using tools. The formal specification is later used as a model for the implementation [7][8]. Basically, the flowcharts and structural diagrams that are used in conventional software development are a bit like those formal specification. The difference with the FDT’s is that there are tools that can be applied to the specification. During the design of the specification it is possible to simulate the model in order to find errors. This helps the developer to debug the specification.

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

Unfortunately, this transition from validated 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 proven that the resulting final implementation will fulfil the requirements.

In order to produce better code, the number of steps from validated 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 themselves 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.

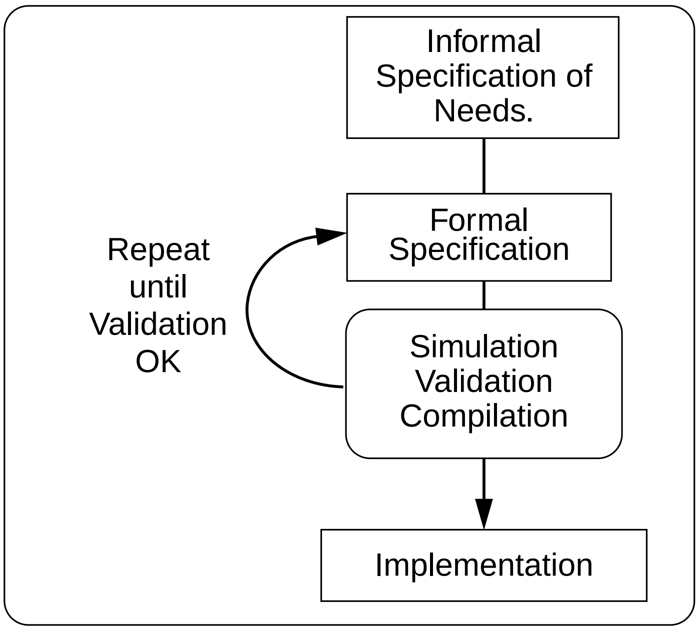

Informal Specification of Needs.
Formal Specification
Repeat until Validation OK
Simulation Validation Compilation
Implementation

Figure 3: Using the validated Specification for the compilation of the Implementation

1.2 Overview over PROMELA and SPIN

1.2.1 History of PROMELA / SPIN

PROMELA (Protocol Meta Language) was developed by Gerard J. Holzmann in the AT&T Bell Laboratories. 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. He presents the complete source code for the tool SPIN and explains the verification techniques that can be applied when using his system.

1.2.2 Architecture of the System

Figure 4 shows an overview of the system. The same tool, SPIN, is used for the simulation and the

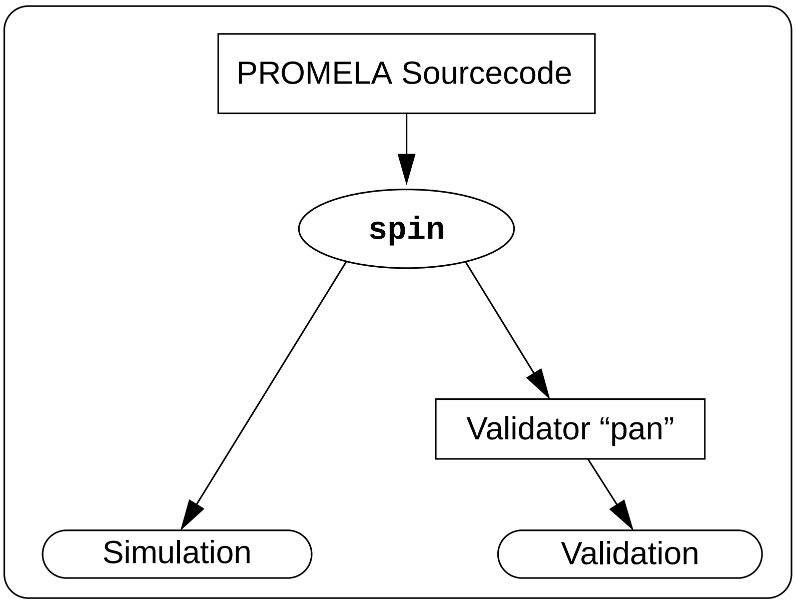


Figure 4: Structure of Holzmann’s PROMELA/SPIN System

validation of the model. For the simulation, the PROMELA source is interpreted. For validation, it is compiled into a state space analyser. This analyser (also called “validator”) is coded in C and consists of a part describing a state machine and of code that systematically walks through a tree of all possible transitions in the state space. It offers multiple algorithms for validation, which are described in detail in the book.

1.3 The Language PROMELA

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. This allows a reduction of the language elements and is the reason for the beautiful simplicity of the language. In addition to that, since the language is based on the theory of finite state machines, it is very easy to understand its concepts for everyone who knows the techniques of state machines.

The most important design goal of PROMELA was the specification of distributed systems. Such systems are represented by sets of concurrent, parallel processes that are able to communicate with each other. The language allows to describe the properties of process prototypes and of global resources such as channels or shared variables that can be used to model the communication between processes.

1.3.1 Basic Object Types

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 using the “`run`” statement. The same process prototype can be invoked multiple times, this makes it possible to build recursive models.

2. **Variables**

   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. A variable definition contains the type of the variable. It is possible to declare arrays of variables. However, there are no structured variables like the "`struct`" in C.

3. **Channels**

   The channels are an exceptional data type. They are essentially finite-length FIFO queues. Like variables, channels can be defined globally for the specification or locally within each proctype. The channels can transmit messages of fixed, predefined types. Channels can be defined as synchronous ("rendez-vous") channels, which allow a synchronisation of two processes. The channels should be used as the main medium for the modelisation of all communication in the specification.

1.3.2 Processes

Figure 5 shows a skeleton of a typical PROMELA program. This program consists of two (empty)

```
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: Skeleton of a PROMELA Program.

process prototype definitions. The "`init`" process is invoked when the program is started and instantiates the two processes “ProcessA“ and “ProcessB“ by executing the “run” statements. Afterwards it writes a message on standard output. As one can see in the example, the process prototype definitions are just descriptions of the structure of a process. In order to actually use those prototypes the “run” statement is necessary.

1.3.3 Variables and Types

As mentioned, variables can be defined either globally or locally for the process prototypes. All definitions that are not inside of a “proctype” or “init” construct are global and can be accessed from all processes. A PROMELA variable has to be of one of the five predefined types, that is: bit, byte, short, int or chan.

In figure 6 some examples for variable definitions are given. Here, the first line declares a variable “a”

```
int a = 1;
int b[10];
byte b;

mtype = {ack, nak, cr, dr, conn, disc}
chan canal = [5] of {chan, mtype};
```

Figure 6: Example for Variable and Channel Declarations

which is initialised with the value 1. The second line serves to declare an array of 10 integers. The third declares the variable b as byte and does not initialise its value. The following two lines are an example for the definition of a channel and are explained in the next paragraph.

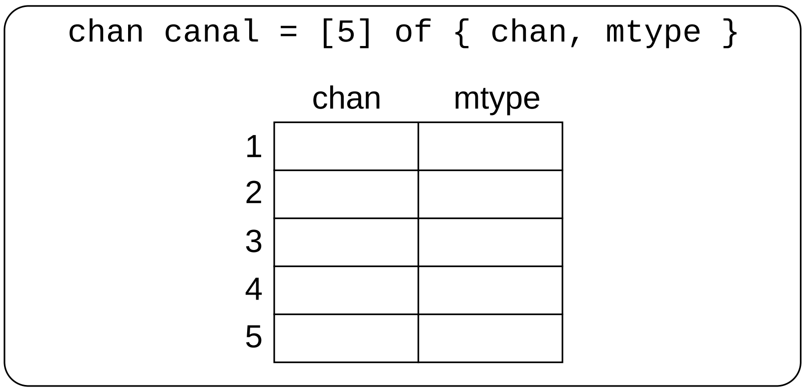
1.3.4 Channels

A channel declaration defines the data types of the messages that can be transmitted over a channel. The data types may contain channel names, this can be very useful for the specification of recursive models.

Channels may either be defined as synchronous (“rendez-vous”) or as asynchronous. Synchronous channels are declared by setting the length of the channels FIFO queue as zero and can be used to synchronise two processes. A write access to a synchronous channel blocks the writing process until a corresponding read access is executed by another process or vice versa.

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.

In figure 6 the line starting with “`chan`” defines a communication channel whose name is “`canal`” and whose FIFO queue contains five elements. The messages transmitted over this channel have to consist of a channel name (type “`chan`”) and an identifier (type “`mtype`”). Figure 7 shows what the created data structure looks like.



```
chan canal = [5] of { chan, mtype }
```

Figure 7: Corresponding FIFO Queue

The possible identifiers are defined in the “`mtype`” statement. In this example, this can be “`ack`”, “`nak`”, “`cr`”, “`dr`”, “`conn`” and “`disc`”. Those message identifiers are internally represented by unique, consecutive numbers. In the example, all “`ack`”s in the source will be replaced by “`1`”, all “`nak`” by “`2`” and so on. In each PROMELA specification there has to be only one “`mtype`” definition. Since one could also just write numbers instead of the identifiers, it is not really necessary to use the “`mtype`” statement. However, using identifiers makes the specification much easier to read and reduces the possibilities for design errors.

All PROMELA source codes are fed to the standard C preprocessor "`cpp`" before they are processed by SPIN. Therefore, all C preprocessor directives like "`#define`", "`#include`", "`#ifdef`", ... can be used. Using "`#define`" statements in specifications that use many different message identifiers, makes them eventually easier to read.

1.3.5 Control Structures

The language PROMELA is very close to automata theory. In fact, the language just offers the means to describe finite automata in a computer language. In the next paragraphs we will see how the properties of finite automatas can be translated into PROMELA specifications.

1.3.5.1 Executability

One key concept of the PROMELA language is the *executability* of statements. In PROMELA, there is no difference between conditions and instructions. If a condition is true, it is executable. Equally an instruction can or cannot be executable. For example a receive operation on an empty channel is not executable. A non-executable transition might later get executable, for example if the contents of the channel changes. In fact, a condition as well as a statement has to be seen as the analogon of a transition in the finite state machine model. If a transition between two states is executable, the state machine can move from one state to the other one. If such a transition changes the state variables (which might also be FIFO queues) of the machine, one would have to call it an instruction. If no variables etc. are changed, it could be called an expression.

For linear sequences of transitions it is easy to translate this into a computer language. In PROMELA, statements and expressions are separated by "`;`" (the semicolon) or "`->`". Both of these constructs have exactly the same meaning, which is that the statements that are separated are to be executed sequentially. In case of branches in the control flow a bit more complicated constructs are needed, as we will see in the following paragraphs.

1.3.5.2 Non-Determinism

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 8 shows a state machine with a non-deterministic choice. If, for example, the value of the vari-

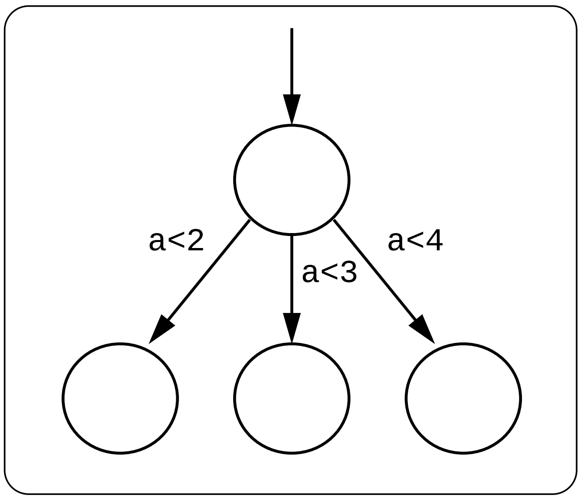


Figure 8: Non-Deterministic Possibilities in a Finite State Machine

able `a` is `1`, all three expressions are “true” and therefore all of them are executable.

1.3.5.3 The “if..fi” Control Structure

The PROMELA control structure that can be used to describe this non-deterministic choice is shown in figure 9. All three transitions that are possible to leave the state depicted as the upper circle are expressed by two colons, followed by one or more expressions. The first expression after the two colons is the one that is tested for executability. If it is executable, the branch may be selected. However, it is possible that there are other executable transitions to leave the state. In this case, any of them can be selected non-deterministically. Once one of them has been chosen, the sequence that fol-

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

Figure 9: Non-Deterministic Possibilities in PROMELA

lows after it is being executed as usual. This sequence can again contain all possible PROMELA language constructs. It has to be emphasised that this sequence can also contain transitions that are not executable, in this case it is still possible for such a sequence to block. The test for executability is only done for the first transition after the two colons.

During the simulation with SPIN the tool will choose between the different executable statements in each “if..fi” statement in a non-deterministic manner. The user can specify if he wants to use random for this choice or if he wants to manually choose the transition to be executed.

In validation, a tree of all possible transitions in the state space is walked through. The analyser writes a “trail” file that is used to log all choices that were made in non-deterministic cases. If errors in the specification are found by the validator, this “trail” file can be used to reconstruct the path that led to the error.

1.3.5.4 The “do..od” Control Structure

For the description of loops, PROMELA offers two possibilities. The first is the infamous “`goto`” Statement. Its syntax is exactly the same as in C, therefore we wont describe it here. The second is the “`do..od`” construct. This one works almost like the “`if..fi`” statement with the difference that after the execution of one of the possible branches, the execution is continued again at the beginning of the “`do..od`” statement. The loop can only be left by a “`break`” or a “`goto`” statement in one of the branches. Figure 10 and figure 11 show an example for a loop that counts from one to four.

```
a=1;
do
:: (a<4) -> a++ -> printf(“%d\n”,a)
:: (a>3) -> break
od
```

Figure 10: PROMELA Source for the “do..od” Loop

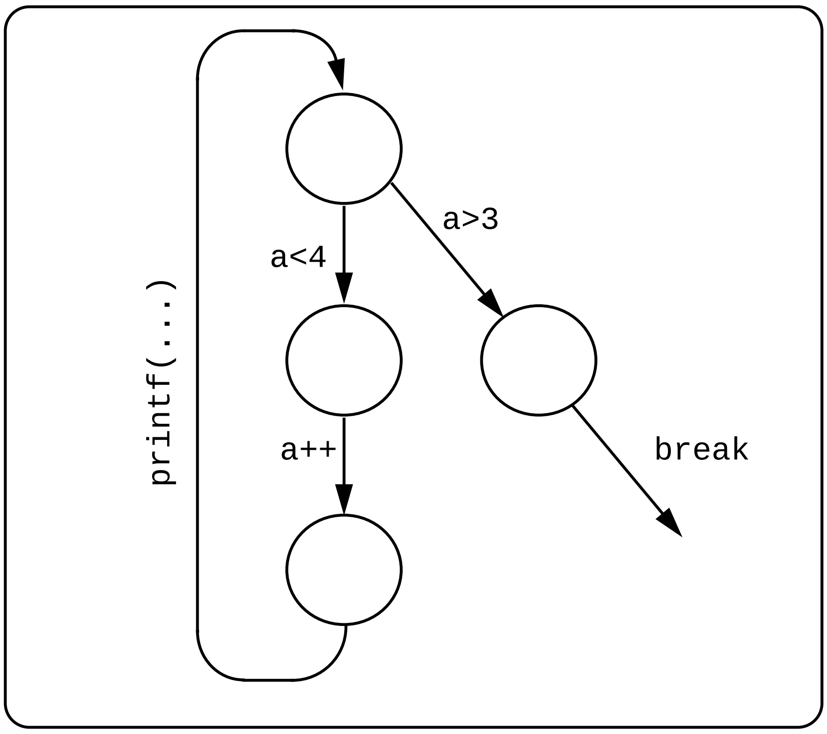


Figure 11: State Machine for the “do..od” Loop

1.4 The Tool SPIN

The ultimate goal of SPIN and indeed of all testing or validation methodology is to demonstrate, with a certain 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?

Since PROMELA/SPIN is not the first and not the only formal description technique that is available, these questions in fact contain the more important one: When and why should one use PROMELA/ SPIN and when are other FDT's to be preferred? And in specific cases: Is it possible to use PROMELA/SPIN for this application?

1.4.1 Possible Verification Criteria

If a software is to be tested at first one should specify for what kind of properties it should be tested. Is it to be checked for calculating wrong results, for not behaving as expected, or simply to be checked if it crashes or hangs? When using SPIN for verification, the following criteria can be applied to the specification:

- 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 (“*unused code*”)?
- Does the system satisfy a user-supplied *temporal logic formula*?

Some of these criteria require that the programmer adds code that contains the details of the specific question that is to be verified, others, like the check for deadlocks, just can be verified by setting the appropriate flag when using the validator.

1.4.2 Degree of Confidence

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. Since the sourcecode is available and documented, it is quite likely that most errors in the analyser itself are already found.

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 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. This information helps to estimate the degree of confidence that one can have in a specific validation.

1.4.3 Cost in Physical and Human Resources

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. For exhaustive validations, the memory usage is still quite high. A larger number of states in a specification results in an exponential increase of the size of the state space (“state space explosion”). A solution to this problem can be to define sequences of statements as atomic, thereby reducing the number of states for this sequence to one. An effort to do this automatically is the “Reactive SPIN” preprocessor by Elie Najm and Frank Olsen [17].

Because PROMELA is easy to learn, the cost in terms of human resources is not too high. On the other hand, it may take a longer time to design a specification than it does with other, more sophisticated FDT’s. This surely depends on the type of the problem.

One advantage of PROMELA is that since its closeness to finite state machines, which makes it relatively easy for a programmer to translate the PROMELA specification into an implementation. With other, more abstract FDT’s this translation can be much more difficult.

2 Design of the Compiler

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 12.

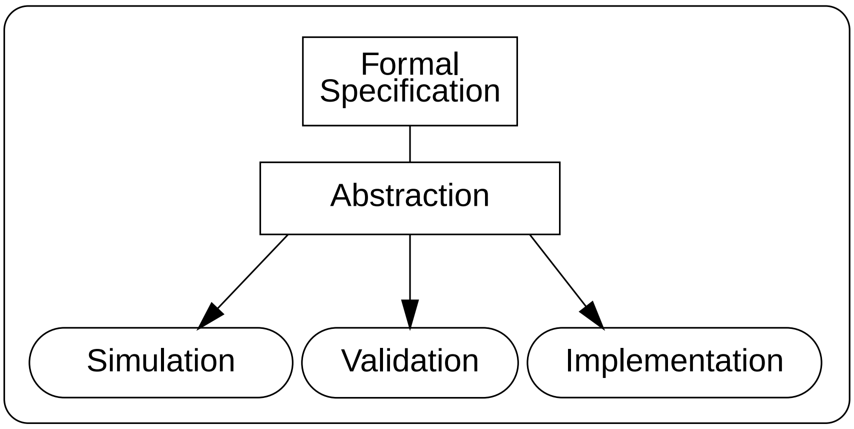


Figure 12: Standard Abstract Tree for FDT Tools

Usually different tools are used for simulation, validation and creation of the implementation. In Holzmann’s PROMELA/SPIN the same tool, SPIN, is used for the simulation and for the creation of the validator. Nevertheless even with PROMELA/SPIN - although it is the same tool that is used - the code that deals with the simulation is quite different from the code for the generation of the validator. As mentioned earlier the PROMELA code is interpreted for the simulation and compiled into the state space analyser “`pan`” for the validation. It is obvious that the code used for simulation and validation is quite different. It was up to Holzmann to take care that the semantics of PROMELA are interpreted exactly the same way during simulation and validation.

This is also valid for the generation of the implementation. The further we leave the existing abstractions that are used for simulation and validation, the larger the probability will be that we introduce differences in the interpretation of the semantics.

The first effort to generate C code from PROMELA specifications was made by E. Moreau and J. Paillet in 1994 [3]. They wrote a totally independent tool to translate PROMELA into C. However, they did not use any part of the existing tools. Their resulting compiler is a bit unsatisfying, because the generated executables do not behave exactly as expected from the simulation. The reason for this is that by rewriting their tool from scratch they had many possibilities to interpret the PROMELA semantics. Unfortunately, since they also rewrote the grammar definitions, they also had possibilities to interpret the syntax differently, therefore it is not even possible to use all PROMELA sourcefiles that were successfully simulated and validated without modifying them.

2.1 Reusing existing Code

In Figure 13 an overview of the data dependencies using SPIN is shown. In order to keep as close as possible to the existing code, it seems to be a good possibility to choose the code that was generated for the validation of the PROMELA model as a starting point for the generation of the implementation. However this makes it necessary to go into the internal data structures of the validator in order to reuse parts of them.

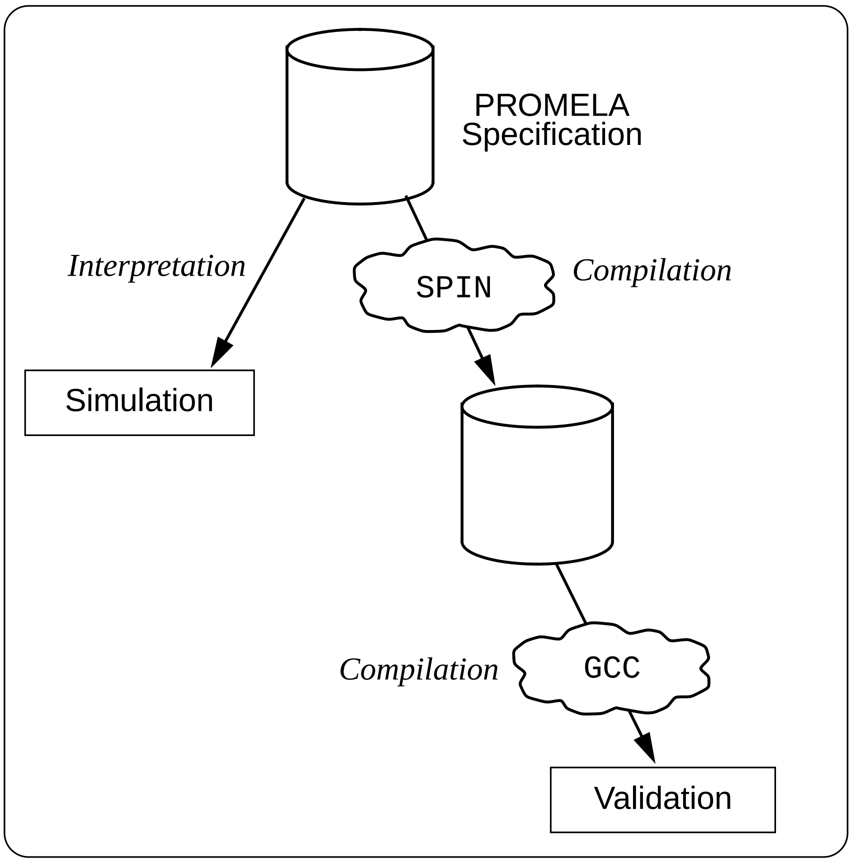


Figure 13: Data Dependencies using SPIN

Figure 14 shows the structure of the validator that SPIN can generate. It is produced as C source code and contains the following five files:

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`”.

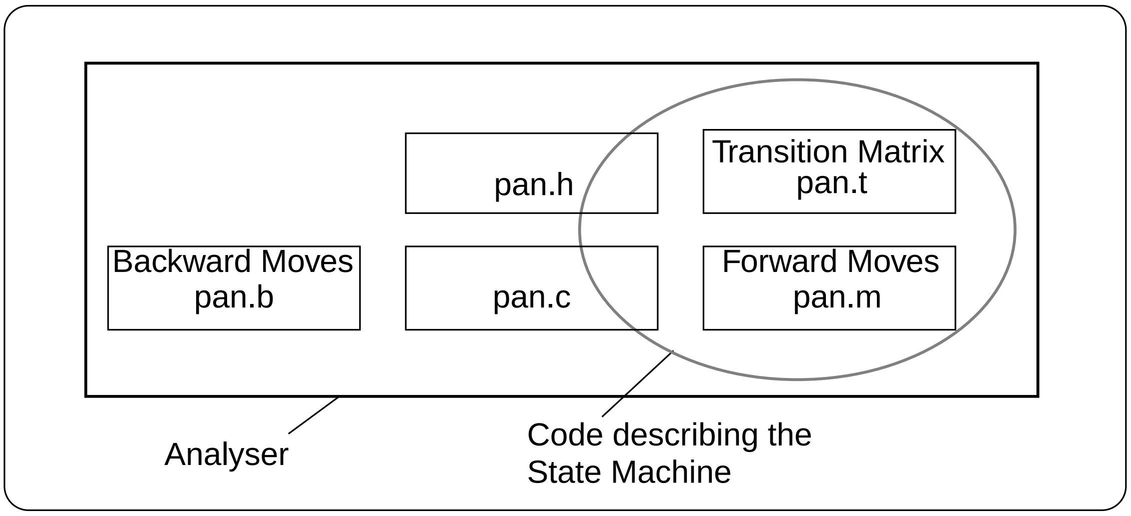


Figure 14: Overview of the Code that SPIN produces when called with the Option “-a”.

Hence SPIN already generated the state machine corresponding to the PROMELA source, we decided to use it as starting point for the generation of the implementation.

However, we do not need to use all of the analysers code. For the execution of the state machine, we are only interested in the transition matrix (“`pan.t`”) and the forward moves (“`pan.m`”). We decided to go a little bit further and to use also some fragments of the file "`pan.c`", the main source file of the validator itself.

In order to create an implementation that uses this state machine, some additional code is needed. In the first place this is some kind of scheduler that chooses the executable transitions and executes the corresponding moves. Additionally, for the development of protocols possibilities for external communications and real-time timers have to be added. Figure 15 shows the design of our extended SPIN environment.

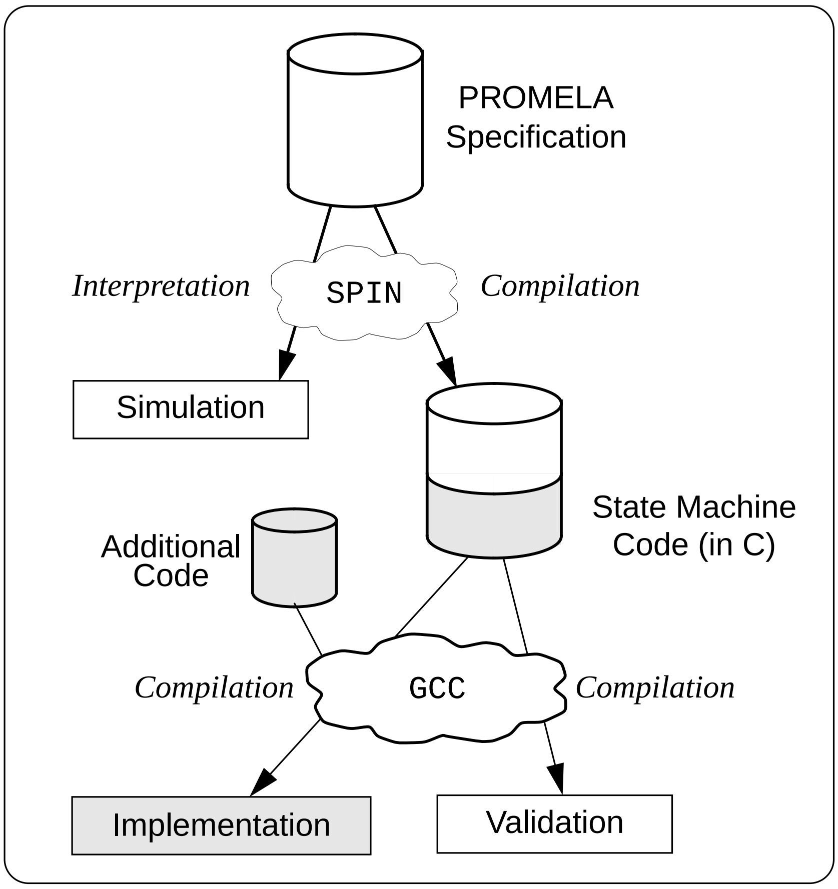


Figure 15: Extending the SPIN System for the generation of Implementations

So our contribution is no separate tool for compilation but in fact consists of modifications to the existing SPIN tool and some additional pieces of code that are needed to run the heart of the analyser: the state machine.

The state machine plus the additional C code for the implementation are compiled into a single UNIX program. Thus, multiple proctypes that are designed to run in parallel will be executed in the same UNIX process. The task of the scheduler is to switch control between the proctypes inside this UNIX process. Figure 16 shows the structure of the UNIX process that is produced as implementation of the

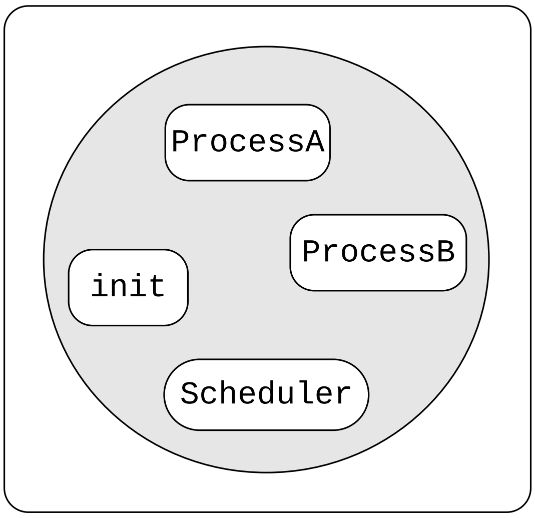


Figure 16: The resulting UNIX Process

model in Figure 5 on page 13.

For the scheduler completely new, additional code is needed. We generate its source code as a separate file which is independent of the PROMELA specification and looks always the same. In the following section, we will describe its design in detail.

2.2 Scheduling

2.2.1 Branch Selection in Non-Deterministic Cases

The most important element that distinguishes PROMELA from other languages is the non-determinism. In Figure 8 and Figure 9 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 risk. For example any of the following strategies could be used for the design of the scheduler:

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 continues with 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.

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.

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

In the current version of the scheduler the first strategy is used. This is very useful for debugging purposes because of the lack of the random element. However, for future versions we also plan to implement the third algorithm because it is closer to the simulation and more suitable for verifying the specification by using the implementation.

2.2.2 Avoiding "Busy Waiting"

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 the same UNIX process then it is not necessary to check continuously whether a transition has become executable ("busy waiting"). The only types of events that could change this state of the UNIX process are:

- External messages from other processes or
- Timeout messages from the timer mechanism

Therefore - in this cases - it is possible to block the entire UNIX process (using the "`select`" system call) 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.

2.2.3 Atomic Sequences

A special case that has to be taken into account for the design of the scheduler are the atomic sequences. In PROMELA, a sequence of statements may be defined as atomar by encapsulating it into an "`atomic{...}`" environment. Such a 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 between processes/proctypes.

So far this is not too hard to implement into the scheduler. However, problems arise when an atomic sequence blocks, for example if it waits for data from a channel that can only be sent by another proctype in the same model. In the first versions of SPIN this was simply forbidden at all. Starting with version 2.0, it is legitimate for an atomic sequence to block. In this case it should - since Version 2.0 - be allowed to switch to another proctype.

This is a real difficult point for the design of the implementation. If we want to add the possibilities for external communications, we can no longer know if an atomic sequence that waits for a message on a channel is blocked or if it is still possible that the external client will send a message and thereby makes it possible that the atomic sequence continues.

We decided not to change the proctype in case of blocking atomic sequences. We think that this is more adequate for the generation of implementations. However, if a proctype switch should be allowed in a certain situation, this has to be specified in the PROMELA source by not putting this transition inside an atomic sequence. 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. This is one of the major drawbacks of PROMELA as a formal description technique today. The semantics are not yet exactly specified - this makes it not only more difficult to use the language but also possible to interpret the model. In fact, this is not only a problem for an automatic compiler that generates an implementation but also for the programmer that manually translates the specification.

2.2.4 Priorities

In Version 2.5 of SPIN, Holzmann added a mechanism to define process priorities for use during random simulations[1].This was implemented to facilitate the debugging of the model in the simulation. For the validation the priorities are useless because all possible cases have to be taken into account and the priorities just have an influence on the probabilities for certain transitions.

For the implementation, the priorities could be used. However since the validation was already made for all possibilities, it is not really necessary to support them. Therefore our scheduler does not do so yet. Maybe they could help for the fine tuning of generated implementations. So this is still an open field for future extensions.

1. See section 2.3.1 in [4]

3 Extensions for Protocol Implementations

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. The original goal Holzmann had in mind when designing it was not to create a language that can be used to describe protocols *in their details* but to provide the means to create an *abstract model* of a protocol. Therefore, the language lacks constructs that are necessary in order to create final implementations. In this chapter, we will describe the extensions we found necessary for the development of protocol implementations.

3.1 Timer Mechanism

A protocol implementation cannot be imagined without taking time into account. The transmission of a message over a real existent communication channel always takes time because the information cannot travel faster than the speed of light. Additionally, for every message that is sent and is to be acknowledged, a timeout value is necessary. Since the original "`timeout`" statement in PROMELA has initially been designed only to provide means to specify an escape from deadlocks, 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 at all for implementations.

A solution to this problem could be to introduce a parameter for the "`timeout`" statement [3]. 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 itself would be necessary and the modified sources would no longer be usable for simulation or validation with not-extended versions of SPIN.

Since we did not want to change the syntax of PROMELA, we searched for another possibility. Our implementation provides a timer mechanism that uses message queues. This mechanism is close to the one of SDL, where timers basically also are described using messages.

For the communication with the timeout mechanism, which is implemented in the runtime scheduler, the following three message channels 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 = [5] 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. The length of the queue has to be long enough to hold all messages that could come from the timer mechanism.

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.

The names of those channels are reserved and must not be used for other purposes. In recursive proctypes that use the timer mechanism, it is advisable to use the “`_pid`” Variable[1] to compute a unique timer identifier (if a “local” timer is required).

To validate a model that uses those timer queues, an additional PROMELA proctype can be added to the specification that describes the timer mechanism in the scheduler, i.e. receives and sends messages via the timer channels. An example how this can be done is given in Section 4.2.

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]).

3.2 External Communications

Figure 17 shows the well known ISO model which is used for protocol design. Assume that we want to specify the communications between layer `N` and `N-1`, as marked in the figure. For simulation and validation we would create a PROMELA sourcefile containing two proctypes, one for each protocol entity, and two globally defined channels. We could then simulate or validate the specification of the whole system with SPIN and check if it is free of errors.

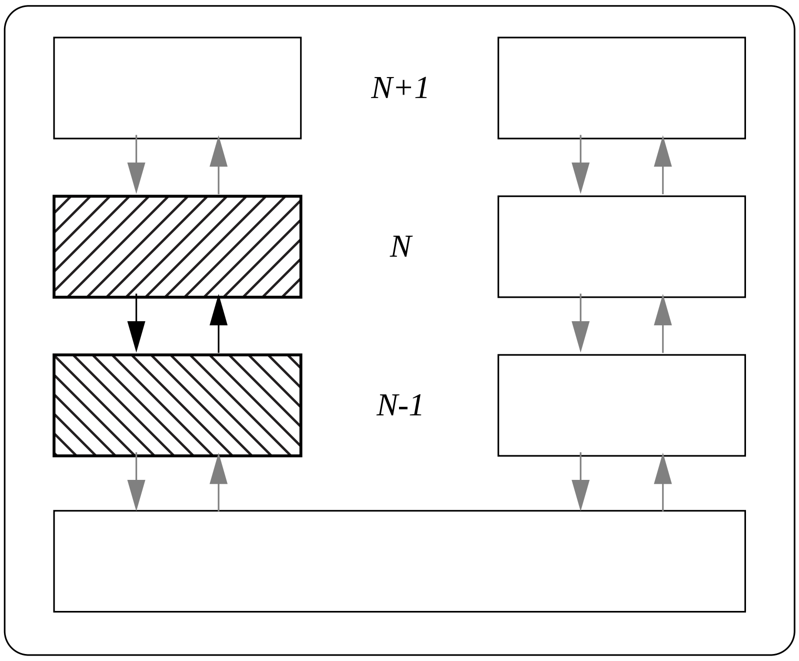


Figure 17: Two communicating Protocol Entities

In order to create an implementation of these protocol entities we do however need to divide the specification into two parts, because we do now want to create two separate UNIX processes. Of course it would be possible to create an implementation of the two protocol entities in one single UNIX process, but it seems quite useless to create an implementation of a protocol that is not able to communicate with its environment.

For the two protocol entities from figure 17 one could for example want to create a scenario with two UNIX processes like depicted in figure 18. In validation and simulation, this scenario has to be specified in one PROMELA sourcefile using two proctypes (one for each protocol instance) and two globally defined communication channels for their interconnection. For the creation of an implementation those two proctypes have to be compiled into two separate UNIX processes, the only thing that has to be common for those two UNIX processes are the globally defined channels. The separation of the proctypes for the two UNIX processes can be achieved using preprocessor directives in the PROMELA sourcecode. For the communications between the two UNIX processes, we do however need some extensions:

- Means to declare a channel as “extern” in the specification
- Some kind of communication mechanism on the UNIX layer that keeps care of the interconnection of the UNIX processes

3.2.1 Defining External Channels

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 all versions of 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 shown in the example in Section 4.2.1

The external channels are used exactly like the internal ones. The only difference is that in the implementation a message sent to an external channel is stored in the communication server of the created client-server model. In the next section we explain how we designed this client-server architecture.

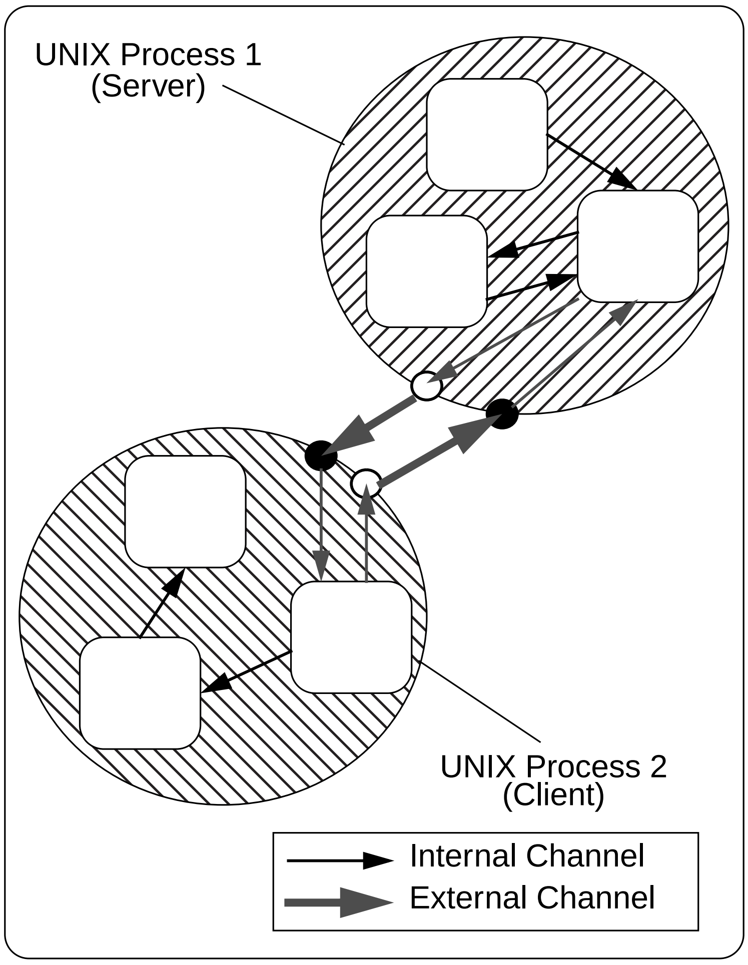


Figure 18: How external Channels can be used to connect UNIX Processes

3.2.2 AF_UNIX Sockets

On the UNIX layer, we decided to use sockets for the interconnection of the processes. In the current version, all UNIX processes have to run on the same machine. For their interconnection, we use the UNIX domain sockets (“AF_UNIX”). The UNIX domain sockets have the advantage that they can easily be replaced by internet domain sockets (“AF_INET”), which then allow a distribution over multiple machines on a network.

3.2.3 Client/Server Model

We have chosen a client-server architecture for the interconnection of the processes because we did not want to limit the number of connected processes to two. In a distributed implementation, there always has to be one server. The number of clients is only limited by the size of the data structures in the generated code. 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. once one UNIX process has read from a channel, thereafter no other UNIX process is granted read access to the same channel[1]. This reduces the communication between the processes and makes them much faster. Nevertheless, multiple proctypes are allowed to write into the same channel queue.

Upon compilation of a PROMELA model, one can specify whether the compiled UNIX process should be a 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 (globally defined variables). We think that all communication can be specified using messages. Since the communication over the network between the two UNIX processes has to be implemented using messages anyway, it does not make much sense to put much effort in an implementation of shared memory between those processes.

1. The reason for this is that a read access always contains of a test for executability of the read statement which is then eventually followed by the execution of the read statement. If multiple processes would try to access the same channel for reading at the same time, it would be necessary to make the test for executability and the real read access atomar in the processes. This would make huge changes to the existing code necessary, therefore it was not implemented.

4 Incompatibilities

The current version of the extended SPIN environment still lacks some features for the generation of implementations. Most of them concern external communications, but some of them are also an aspect for the generation of stand-alone implementations. The missing features documented here are features that can be added in a later release of the compiler since most of them are not relevant for the design of the scheduler.

4.1 External Channels

The most important features concerning the external channels that are not yet implemented are:

- Sorted Send and Random Receive Operations
- Synchronous External Channels

The “Sorted Send” and “Random Receive” operations can however be “simulated” in the meantime by receiving all the data from an external channel into a local channel and afterwards doing the “sorted” operations on this channel.

As for the synchronous external channels, there is currently no possibility to replace them. On the other hand, for the design of “realistic” models, external synchronous channels seem to be a bit academic, because they can not really physically exist. Two programs that run on two separated machines can never really do two operations exactly synchronous. Therefore for most models the synchronous external channels should not really be necessary. The design of our client/server model however allows to add them in a later revision of the compiler.

4.2 New Statements from SPIN 2.0

In version 2.0 Holzmann added some new features to SPIN and PROMELA. One of those are the “sorted send” (`!!`) and “random receive” (`??`) operations we just mentioned. As we said, those are still supported for stand-alone implementations but not for distributed implementations. The reason why most of the new statements do work for stand-alone implementations is again that our generated implementation uses just the same code as the validator. However this is not possible for external channels - here we had to rewrite all procedures that are used to access the channel, and this is the reason why some of the statements for external channels are not yet implemented.

Another case were the code from the analyser is not usable are the statements that concern the scheduling of the next transition. In the old version of SPIN this was only the “`atomic`” statement, however in the new version Holzmann added an “`d_step`” construct which can be used to simplify the model that is used in validation.

A statement that was also introduced with version 2.0 is the “`else`” statement. This construct is currently supported. However it also has an influence on the design of the schedulers code, as we saw in Section 2.2.

In general, all constructs that have an influence on the scheduling will make changes to our scheduler necessary. All other changements and additions to the language that were made should not have much influence on the implementations.

4 Usage of the Compiler

This chapter describes the usage of the extended SPIN environment and shows how an implementation can be created by giving an example session.

The following possibilities were added for the usage of the tool SPIN:

- One can compile the PROMELA specification into a single, stand-alone implementation coded in one UNIX program.
- The generated program can contain additional server code or
- additional client code, which is used to allow the implementation to communicate with other UNIX processes via sockets.

In order to use those new features of SPIN, the following command line arguments can be used:

TABLE 1. New command line options for the extended SPIN tool

In order to...	**... use argument**
create a stand alone UNIX process	-c
create a UNIX program which contains server code	-S
create a UNIX program which contains client code	-C

Using the “`-c`” option to compile the PROMELA specification, the following files are generated:

- “`com.c`” contains the main procedure for the implementation, the scheduler and the timer mechanism. Its contents is basically always the same and does not depend on the PROMELA source.
- “`com.m`” contains the forward moves. It corresponds to “`pan.m`” in the analyser.
- “`com.t`” contains the transition matrix, corresponding to the analysers “`pan.t`”

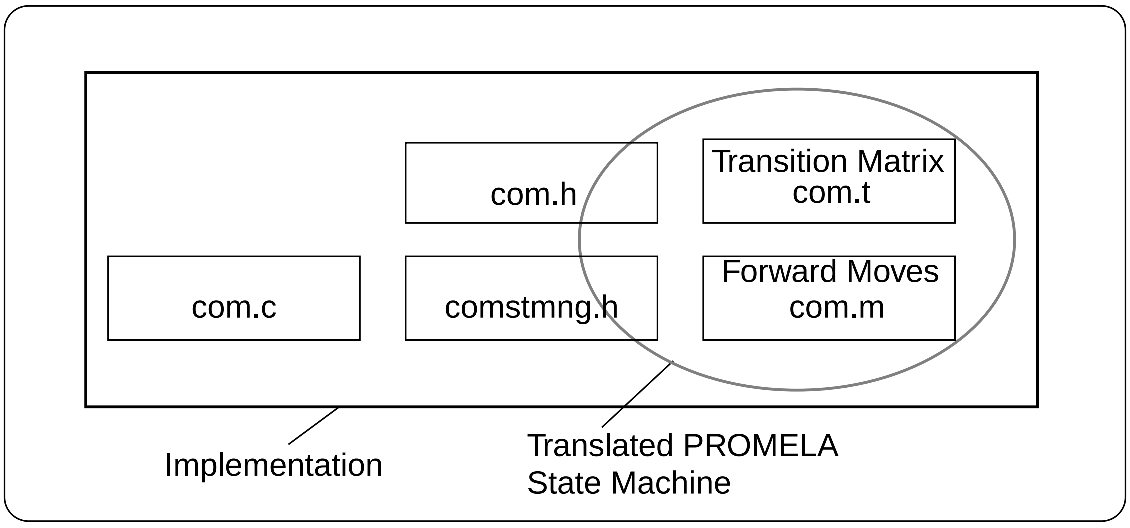


Figure 19: Overview of the Code that SPIN produces when called with the Option "-c".

• "`com.h`" is the counterpart of "`pan.h`" and contains various data structure definitions.

• "comstmng.h" contains routines from the analyser "`pan`" that are reused for the generation of the implementation.

Figure 19 depicts the structure of the generated UNIX program. The files that are highly dependent on the PROMELA source code are the ones within the grey circle, the others are more or less the same for all compiled programs.

By using the "`-C`" or "`-S`" option (in order to generate a client or server program) additional files are produced:

• With "`-C`" in addition to the usual code a file "`comcli.h`" is generated. This file contains the communication routines for the client. Those routines do not depend on the PROMELA source code, therefore the generated code is identical for all compiled clients.

• With “-S” the file “comserv.h”, containing the server communication routines, is generated and included in the implementation code. This file also is not dependent on the PROMELA source and is identical for all generated clients.

Comparing the client/server code with the stand-alone implementation, the only differences that are to mention are the additional files with communication routines and one “#define” statement that is inserted into “com.c” in order to activate those routines. The heart of the implementations, the scheduler and translated state machine, is independent of whether a stand alone program or a distributed implementation is to be generated. Figure 20 and figure 21 show the structures of the UNIX programs for client and server.

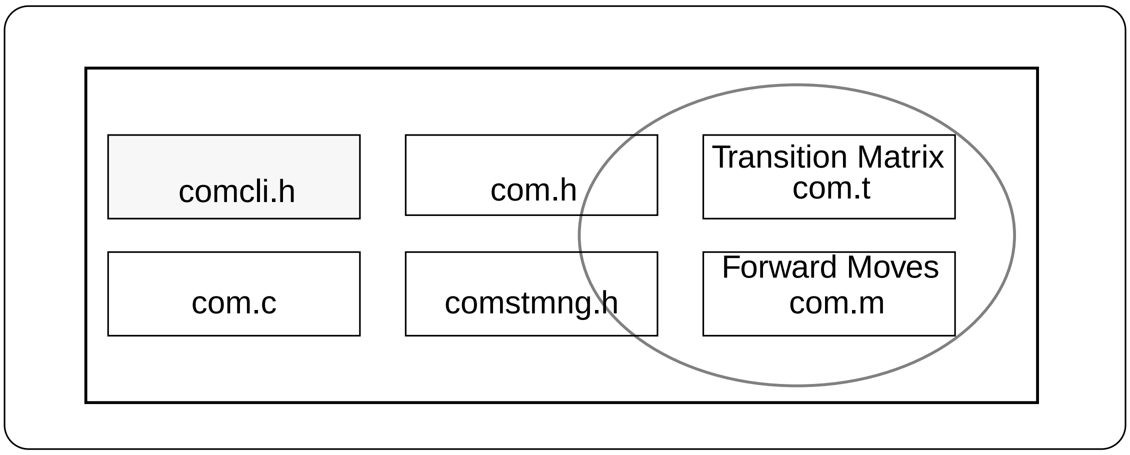


Figure 20: Overview of the Code that SPIN produces when called with the Option “-C”.

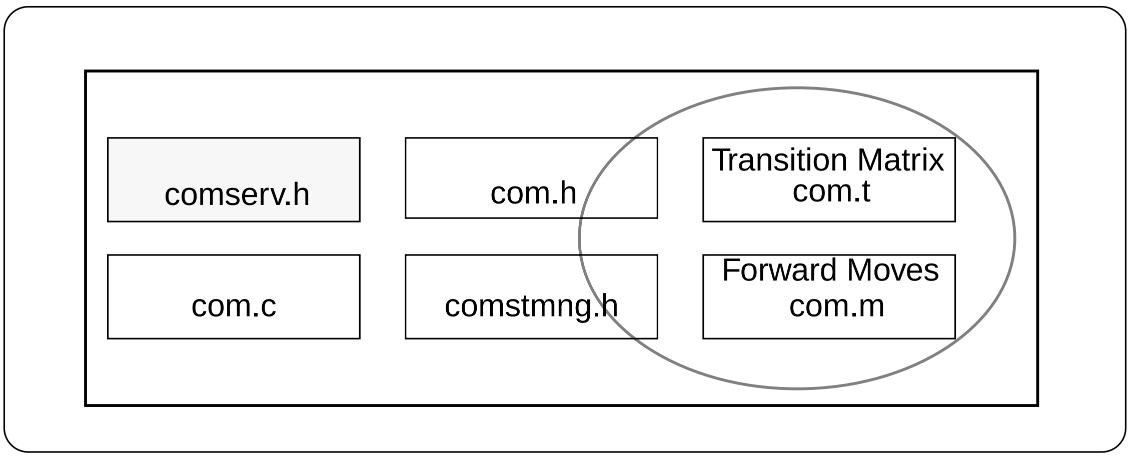


Figure 21: Overview of the Code that SPIN produces when called with the Option “-S”.

4.1 Creating a Stand-Alone Implementation

In this section we will demonstrate how a stand-alone implementation can be created from a PROMELA specification. As an example we specify a counter loop that prints the values from zero to ten on standard output.

4.1.1 Simulation

Figure 22 shows the PROMELA source code for the counter. The first step that we will do after

```
proctype counter()
{
byte cnt=0;

do
 :: printf(“Counter=%d\n”,cnt); cnt++;
    if
      :: (cnt<10) -> skip
      :: (cnt==10) -> goto end_it;
    fi
od;
end_it:
   printf(“End.\n”);
}

init
{
   run counter()
}
```

Figure 22: PROMELA Model for a simple Counter

having specified the counter is to simulate it using SPIN. Figure 23 shows a log of the UNIX session in which this has be done. However, like depicted this example is a bit boring, because there are no errors in the model. Therefore a simulation and validation of this model is quite pointless, because there are no assertions or assumptions in it that are to be verified and it is so simple that the results are obvious. However, we can easily introduce a “bug”. Since the variable `cnt` in the example is of type

```
unix$ spin counter.pr
Counter=0
Counter=1
Counter=2
Counter=3
Counter=4
Counter=5
Counter=6
Counter=7
Counter=8
Counter=9
End.
2 processes created
unix$
```

Figure 23: Sample Simulation Session for the Counter

`byte`, its value has to be in the range from 0 to 255. Setting it to 256 should result in a value of 0 for the variable. So if we change the upper limit for the counter loop to 256, SPIN should be able to detect that at the moment where the variable cnt is increased from 255 to 256 (that is, “increased” to zero), the whole state machine of the program reaches a state that it already was in before. During simulation SPIN then gives us a warning message like depicted in Figure 24.

```
Counter=255
spin: line 9 “counter.pr”, Error: value (256->0 (8)) truncated
in assignment
spin: line 8 “counter.pr”, Error: value (256->0 (8)) truncated
in assignment
Counter=spin: line 6 “counter.pr”, Error: value (256->0 (8))
truncated in assignment
0
spin: line 6 “counter.pr”, Error: value (256->0 (8)) truncated
in assignment
Counter=1
```

Figure 24: Warnings in Simulation Mode

4.1.2 Validation

Next we will try if the analyser generated by SPIN is able to find this type of error. In order to find a non-progress cycle (“livelock”) in validation mode, the “-l” command line option has to be given to the analyser. Figure 25 shows the results that the validator “pan” prints out in this case. As expected, the non-progress cycle is found and an error is reported.

```
unix$ spin -a counter.pr
unix$ gcc pan.c -o pan
unix$ ./pan -l
Counter=0
Counter=1
Counter=2
...
...
Counter=255
pan: non-progress cycle (at depth 1024)
pan: wrote counter.pr.trail
(Spin Version 2.7.5 -- 7 October 1995)
Warning: Search not completed

Full statespace search for:
 never-claim - (none specified)
 assertion violations +
 non-progress cycles + (fairness disabled)
 acceptance cycles - (not selected)
 invalid endstates +

State-vector 16 byte, depth reached 2047, errors: 1
 1025 states, stored (2049 visited)
 1 states, matched
 2050 transitions (= visited+matched)
 0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.265e+06 memory usage (bytes)

unix$
```

Figure 25: Sample Validation Session

4.1.3 Compilation

Now that we have seen how the model can be simulated and validated, we change back the upper bound for the counter to 10, so that simulator and validator do not report any errors. In order to create an implementation, we will use the new "`-c`" option of the extended SPIN tool. This creates a stand-alone UNIX program. Since there is no non-determinism in this specification, the implementation should produce exactly the same output as the simulation did. Figure 26 shows that this is true, the

```
unix$ spin -c counter.pr
unix$ gcc com.c -o com
unix$ ./com
Counter=0
Counter=1
Counter=2
Counter=3
Counter=4
Counter=5
Counter=6
Counter=7
Counter=8
Counter=9
End.
unix$
```

Figure 26: Sample Simulation Session for the Counter

compiled UNIX program behaves just as expected.

If we would have non-deterministic choices in the specification, our implementation would most probably not behave exactly like the simulation, because another scheduler is used to choose between the possible branches, as explained in Section 2.2 on page 29.

4.2 Creating a Distributed Implementation

4.2.1 A PROMELA Specification of the Alternating Bit Protocol

As an example for the compilation of a distributed implementation, we use Lynchs “Alternating Bit Protocol” (ABP) [16]. The “Alternating Bit Protocol” is a very simple protocol that can be used to assure that each message transmitted by the sender is received at least once by the receiver.

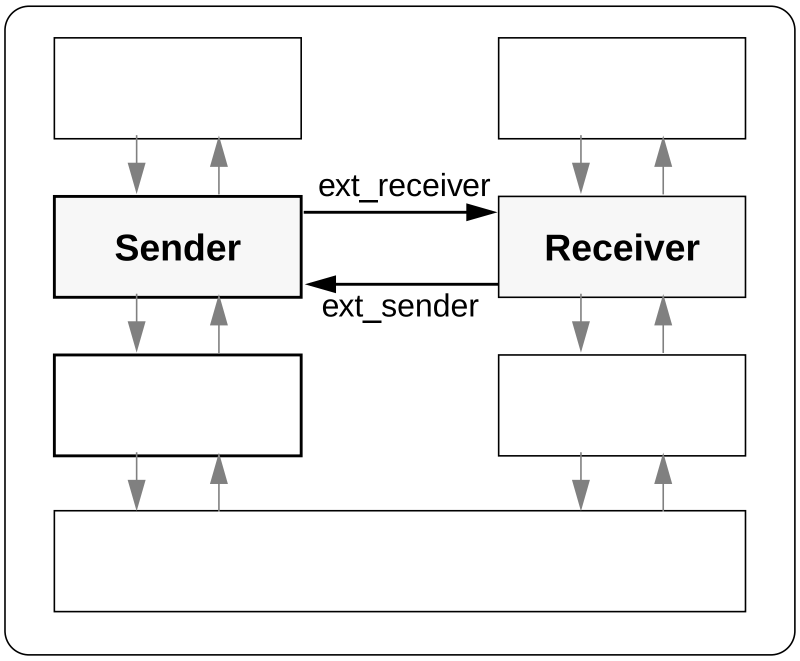


Figure 27: Embedding of Sender and Receiver Entity

Figure 27 shows how the protocol can be positioned in an ISO layer model. The sender and the receiver entity are on the same layer and communicate with each other via the lower layers. However, for the specification of the sender and the receiver, the specification of the lower layers is of no interest for us, so we just describe the interconnection between those two entities by two communication channels, depicted with the two horizontal flashes between the entities in figure 27.

The sender and the receiver were modeled in two PROMELA proctypes shown in figure 28 and figure 29. They are kept in separate files to allow the creation of a distributed environment.

```
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 28: The PROMELA Model for the Sender

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 33). Because the timer mechanism of the runtime scheduler has to be taken into account in validation / simulation, an extra file that describes this mechanism has to be included for validation and simulation. The timer mechanism could be specified like depicted in figure 34. However, this specification of the timer mechanism is a bit academic,

```
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 29: The PROMELA Model for the Receiver

because the "`run`" statement that is used to instantiate new timers will allocate more and more resources during simulation or validations.

For the creation of two separate UNIX processes, two other files are necessary. Those two files are shown in figure 32 (for the sender) and figure 35 (for the receiver). They essentially contain the "`run`" statement that is used in each of the UNIX programs to finally instantiate the process.

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

Figure 30: Includefile "mychans.h" with Definitions of all global Channels

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

init
{
   run sender()
}
```

Figure 32: PROMELA Sourcefile for the Implementation of the Sender

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

Figure 33: PROMELA Sourcefile for Simulation and Validation

Each of the compiled processes may either be the server process or the client process, depending of the choice of the command line arguments when compiling them.

4.2.2 Implementation of the Alternating Bit Protocol

So what can we do with this specification? First let us see how it can be simulated. Figure 36 shows a log of a sample simulation session. Note that the receiver process printed its message stating

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

Figure 31: Includefile “timers.h” for Timer Channel Definitions

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

Figure 34: PROMELA Source "simtimers.h" that describes the new Timers for Simulation / Validation

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

init
{
   run receiver()
}
```

Figure 35: PROMELA Sourcefile for the Implementation of the Receiver

that it has received "`msg1`" before the sender process printed out the message "`MSG1-->`". This is no error, because transmission ("`!`") and "`printf`" statements are not inside an atomic construct, and therefore the simulation scheduler is allowed to switch between processes between the execution of those two statements.

The validation of the specification is also done like usual. As mentioned before, the "`run`" statements in the file "`simtimers.h`" will have the result that the specification is not suitable for exhaustive validation. The proposed increase of "`VECTORSZ`" won't change this. The only possibility to allow exhaustive validation is to change the specification of the timers. However this would be too much to do it here, so we won't care about the validation of this model.

```
unix$ spin simval.pr
      ->msg1
MSG1-->
      <-ACK1
ack1<-
MSG0-->
      ->msg0
      <-ACK0
ack0<-
MSG1-->
      (lost)
timeout1

...
```

Figure 36: Sample Simulation Session for the ABP

```
unix$ spin -a simval.pr
unix$ gcc pan.c -o pan
unix$ ./pan
      ->msg1
      <-ACK1
MSG1-->
ack1<-
      ->msg0
      -ACK0

...

pan: VECTORSZ is too small, edit pan.h (at depth 1685)
pan: wrote simval.pr.trail
```

Figure 37: Sample Validation Session for the Counter

Now comes the interesting part which differs from the previous chapter. For the generation of the distributed implementation, we first have to choose which of the two processes should be the client and which should be the server process. In this example it seems to make more sense to make the sender the server process, since the client's task is rather simple (the client just echoes the acknowledgments). So we will use the “`-S`” command line argument to compile the sender and the “`-C`” argument to compile the receiver. Figure 38 shows a log of the UNIX session in which the two programs

```
unix$ spin -S sender.pr
unix$ gcc com.c -o send
unix$ spin -C receiver.pr
unix$ gcc com.c -o recv
unix$ ./send
Creating new /tmp/.spinsocket File
Server ready for connections.
MSG1-->
```

Figure 38: Compilation of Client and Server

were compiled. After launching the server (the sender), it creates a special file “`/tmp/.spinsocket`” which can then be used by the client to connect the server. The server always has to be started before the client. When the server is running, the client process can be started (in another window, shell or whatsoever) and will have access to the “`ext_`” channels of the server. Upon each new connection or disconnection of a client, the server process will print out a message on standard output.

It has to be taken care that the server process does not finish before the client process is terminated, otherwise all further accesses to external channels by the client process will have a “broken pipe” error as result.

5 Examples

In order to test our compiler, we looked for some typical example applications. The first one we tried to implement was the “Alternating Bit Protocol”, as described in Section 4.2 on page 49ff. This is one of the most basic protocols we found and was used to verify if the compiler is usable for the generation of protocol implementations.

Then we looked for a more complicated example, that really requires a verified implementation. We found it in the “Steam Boiler Control Specification Problem” [18][19]. However, this problem is quite big and we found that it takes quite long to solve it with PROMELA. So in this chapter we will describe the solution to the problem that we specified in order to test our compiler. This solution is not yet a full answer to the questions posed by the problem, it still misses important aspects. It does however show that our extended SPIN environment can be used to deal with the problem. There are many other specifications of the steam boiler control, among them is also one in PROMELA by G. Duval and T. Cattel of the EPFL (Switzerland) [20] which we successfully compiled into a stand-alone implementation. Since the sourcecode of this specification is not very easy to read, we did nevertheless decide to rewrite our own specification, which was aimed for beeing implemented at a later time from the beginnning and is therefore much more suitable, especially for the design of a distributed implementation. However it is not as complete as the specification from Duval and Cattel.

5.1 The Steam Boiler Problem

5.1.1 Introduction to the Problem

The “Steam-boiler Control Specification Problem” [18][19] was given to the participants of the Dagstuhl meeting “*Methods for Semantics and Specification*” which was organized by Egon Börger (Pisa) and Hans Langmaack (Kiel) in June 1995.

The specification which was initially published by Jean-Raymond Abrial describes a control program that serves to control the water level in a steam boiler by communicating with a set of physical devices. It is based on a specification by the “Institute for Risk Research” and the “Institut de Protection et de Sureté Nucléaire” and therefore it is very formal and strongly aimed at a particular implementation.

The task of the control program is to maintain the water level in the boiler between the two limits `N1` and `N2`. The level must not pass under/over the limits `M1`/`M2` for more than five seconds, otherwise the boiler can be damaged. Since everyone can imagine what this would mean to a nuclear power plant, it is obvious why it makes sense to try to validate the control program with a formal description technique.

The physical system comprises the following devices:

- The steam-boiler.
- A device to measure the quantity of water in the steam-boiler
- Four pumps to provide the steam-boiler with water
- Four pump controllers to supervise the pumps (one for each pump)
- A device to measure the quantity of steam which comes out of the steam boiler

Examples

- A valve to empty the boiler during the initialization phase
- An emergency-stop switch (“operator desk”)
- A message transmission System

Since the heating of the boiler cannot be influenced by the program, the only way to control the water level in the boiler is by switching pumps on or off (except during the initialization phase, where the boiler can be emptied using the valve). Figure 39 shows the physical structure of the steam boiler.

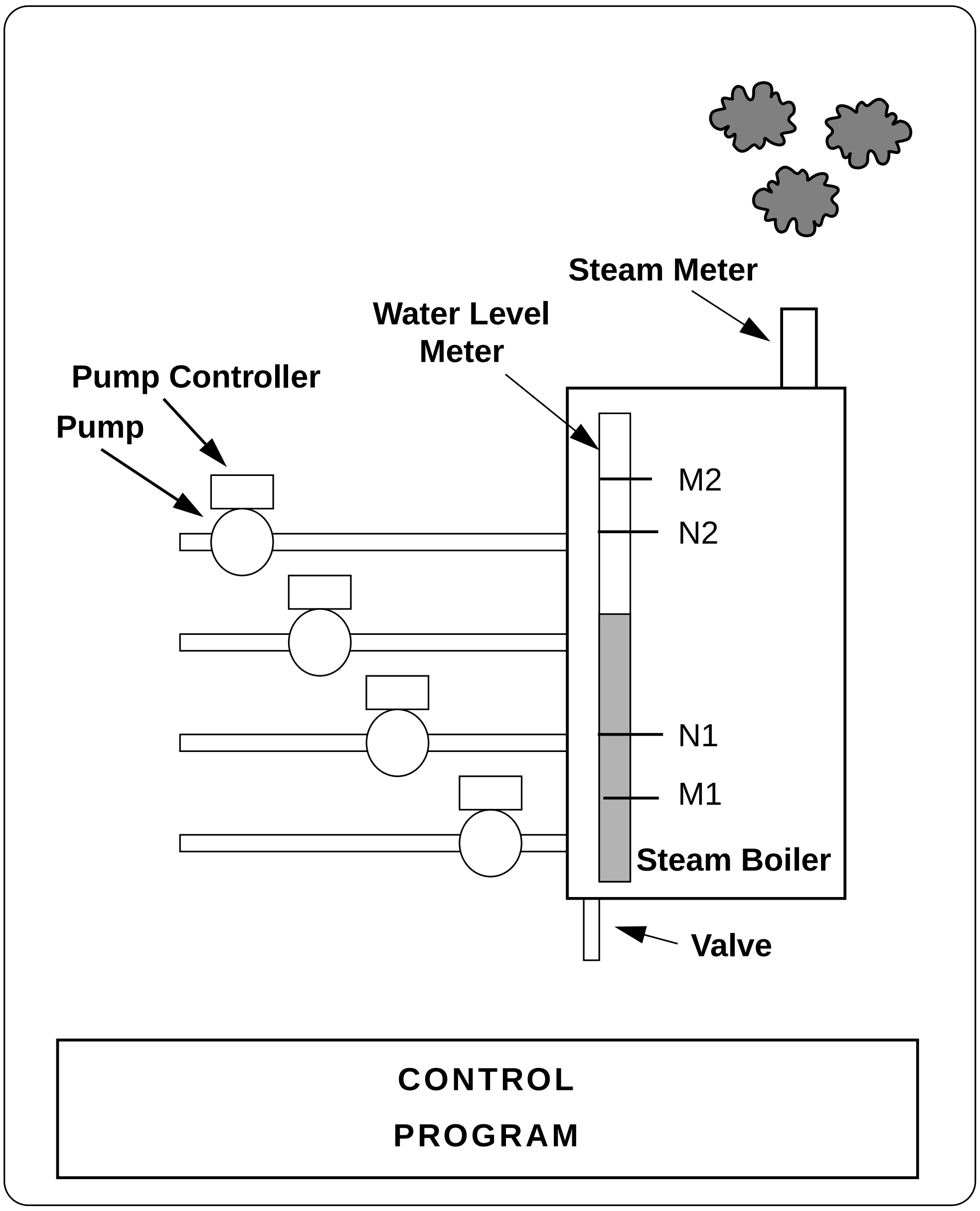


Figure 39: The Physical Structure of the Steam-Boiler System

5.1.2 Structurisation of the Problem

The problem description [18] proposes a structure for the specification of the system by pre-defining a list of messages between the control program and the set of the physical units. For a PROMELA specification, we need to divide the system into several processes that communicate with each other, either via message channels or via shared variables.

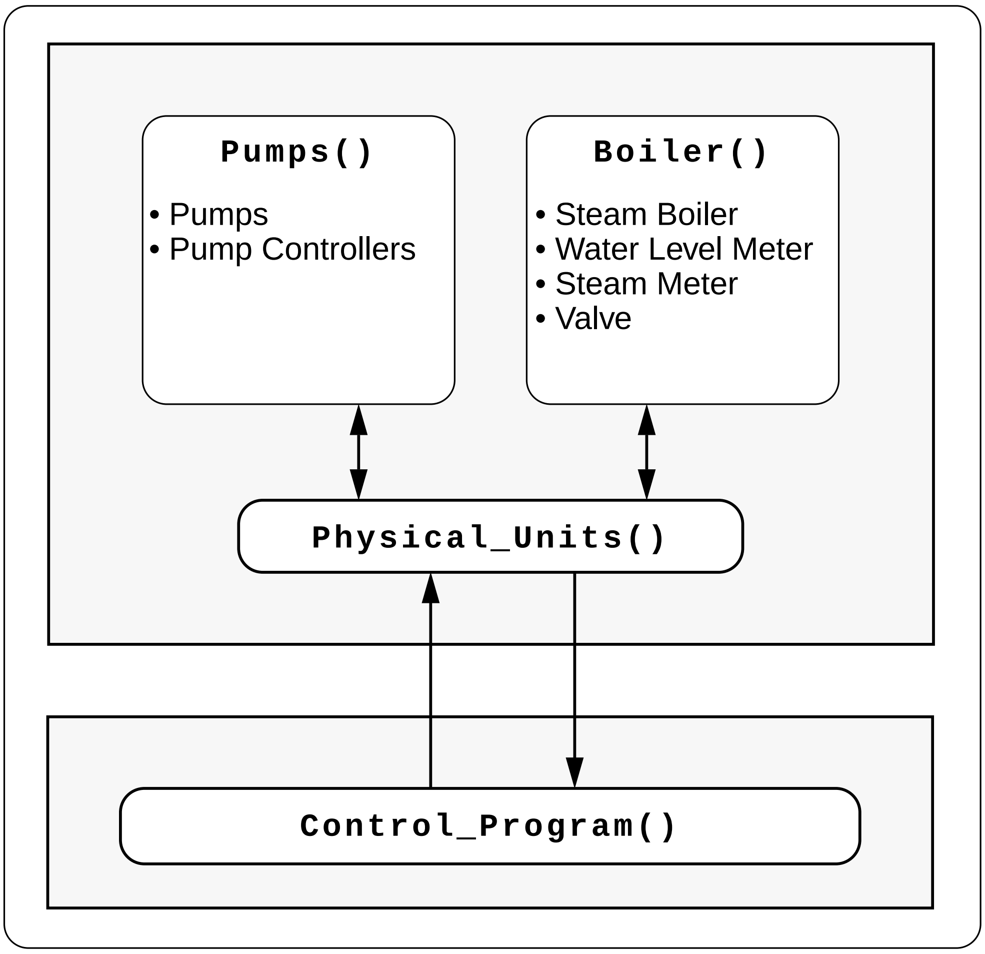


Figure 40: The Logical Structure of the PROMELA Specification for the Steam Boiler

The structure given by the problem description was mostly maintained by introducing a process called "`Physical_Units`" which is the only one that communicates with the control program.

For the physical entities themselves there are separate processes. There is one for the four pumps and pump controllers and one for the boiler including the steam- and water-level sensor and the valve.

All communication in the model is assumed to be *asynchronous*, since all entities run in parallel and there is no information given about the time being used for the transmission of messages between the units. The asynchronous communications are more adequate to an implementation, because in the real physical environment all units run independent from each other.

Figure 40 shows the conceptual structure we used for our PROMELA specification.

5.1.3 Interpretation of the Informal Description

Because the problem description [18] was derived directly from an industrial specification, the text is very informal and there are many aspects in it that can be interpreted. However the main goal for solutions to the problem should be that the solution is usable to “demonstrate that the specification can be refined to an implementation that is close enough to the functional requirements” of the informal specification, so wherever an interpretation is possible, it was tried to make an assumption that seemed realistic.

In particular, the following assumptions have been made:

- Only one pump can be switched on or off in one cycle.
- All communication is asynchronous (takes time)
- A pump needs no time to start pouring water into the boiler.
  Instead of assuming that a pump needs five seconds to start pouring water into the boiler, we assume that the pump starts immediately pouring water into the boiler. To compensate this, the water level is not allowed to go under/over `M1`/`M2` at all.

- In ’Emergency Mode’ it is assumed that the system should try to switch off all physical units as fast as possible and then terminate its execution. Therefore some new messages have been added to signalize the units that they should terminate.

5.1.4 Real-time Dependencies

PROMELA offers no language constructs that would allow a real-time modelisation. The only statement about time that is made in the specification is that a complete control program cycle should take five seconds. In these five seconds the state of the boiler may change continuously. This was simplified in the PROMELA model: The levels in the boiler are only recomputed for whole cycles of five seconds. Thereafter the boiler process is resynchronised with the other processes by sending out a status message and waiting for a synchronisation message.

5.1.5 Structure of the PROMELA Sourcecodes

The sourcecode was structured after the logical structure which we developed in Figure 40. It contains the following code sections:

- A section with “`#define`“ statements (which are used instead of the usual PROMELA “`mtype`“ statement)
- A section with definitions of global variables. These variables are only used as global variables for the different proctypes that describe the physical units. The control program itself has only local variables, all communication between control program and physical units is done by means of communication channels.
- A definition of system constants that describe the physical limits / capacities of the boiler.
- The proctype “`Control_Program()`“ is the main control program itself.
- The proctype “`Pumps()`“ is the proctype that describes the Pumps and the Pump controllers.

- The proctype “`Boiler()`” is used to update the levels in the boiler, steam production etc. It specifies the valve by emptying the boiler instantaneous in case of a “open valve” command. By providing the actual data over water level and steam production it also modelises the water level and steam meter.

- The proctype “`Physical_Units()`” is responsible for the communication with the control program and all proctypes describing the real physical units.

- Finally, “`init()`” is the initial proctype which starts up the control program and the physical units.

5.1.6 Definitions of Global Resources

The following is the definition of the global resources, which is common for the control program and the physical units. In order two create two separate implementations, this file is included into each of the specifications.

```
/* Constants ------------------------------------------------ */
#define OFF 0
#define ON 1
#define OK 2
#define DEFECT 3

/* Communication Channel Declarations ---------------------- */
chan ext_physunits = [4] of { int, int };
chan ext_ctrlp = [4] of { int, int };

/* The following channels are used for the transmission */
/* of Messages with Parameters from the Physical Units to */
/* the Control Program */
chan ext_ctrlp_pumpstate = [2] of { int, int, int, int };
chan ext_ctrlp_pumpctrlstate = [2] of { int, int, int, int };
chan ext_boiler_timer = [1] of { int };
chan ext_boiler_sync = [1] of { int };

/* Message Identifiers ------------------------------------- */
/* Messages between the physunits proctype and the “real” */
/* Components of the Physical Units Segment, that is Pumps, */
/* Boiler, ... */

/* Messages sent from control to physical units: */
```

```
#define physunits_mode 300
#define physunits_program_ready 301
#define physunits_valve 302
#define physunits_open_pump 303
#define physunits_close_pump 304
#define physunits_pump_failure_detection 305
#define physunits_pump_control_failure_detection 306
#define physunits_level_failure_detection 307
#define physunits_steam_failure_detection 308
#define physunits_pump_repaired_acknowledgment 309
#define physunits_pump_control_repaired_acknowledgment 310
#define physunits_level_repaired_acknowledgment 311
#define physunits_steam_repaired_acknowledgment 312

/* Parameters for the ‘physunits_mode’ Messages: */
#define mode_init 10
#define mode_normal 11
#define mode_degraded 12
#define mode_rescue 13
#define mode_emergency 14

/* Messages received by the Program (emitted by the phys. units) */
#define ctrlp_stop 400
#define ctrlp_steam_boiler_waiting 401
#define ctrlp_physical_units_ready 402
#define ctrlp_pump_state -1
#define ctrlp_pump_control_state -1
#define ctrlp_level 403
#define ctrlp_steam 404
#define ctrlp_pump_repaired 405
#define ctrlp_pump_control_repaired 406
#define ctrlp_level_repaired 407
#define ctrlp_steam_repaired 408
#define ctrlp_pump_failure_acknowledgment 409
#define ctrlp_pump_control_failure_acknowledgment 410
#define ctrlp_level_failure_acknowledgment 411
#define ctrlp_steam_outcome_failure_acknowledgment 412
#define ctrlp_end_of_transmission 413

/* System Constants ---------------------------------------- */
#define C 1200 /* Maximal capacity of water in boiler [liters] */
#define M1 100 /* Minimal limit [liters] */
#define N1 400 /* Minimal normal [liters] */
#define N2 600 /* Maximal normal [liters] */
#define M2 900 /* Maximal limit [liters] */
#define W 10 /* Maximum output quantity of steam [liters/second] */
#define U1 2 /* Maximum grad. increase of steam [liters/second^2] */
#define U2 2 /* Maximum grad. decrease of steam [liters/second^2] */
#define P 5 /* Nominal capacity of pump [liters/second] */
```

5.1.7 Source Code for the Control Program

The following is the definition of the process prototype for the control program. As already mentioned, it communicates with the physical units process via channels.

```
proctype Control_Program()
{
 int ps[5]; /* Pumpswitch-states rcvd from Physunits */
 int eps[5]; /* Expected Pumpsw.-states in next Cycle */
 int pumpstate[5]; /* Stored Pumpstates (Defect, OK) */
 int pcs[5]; /* Pump Control States recvd from Physu. */
 int epcs[5]; /* Expected P. Control States in next C. */
 int level,lastlevel;
 int steam,laststeam;
 int stopcounter=0;
 int rec_stop=0;
 int rec_physunits_ready=0;
 int nr, state;
 int waterlevelunit=OK;
 int steamlevelunit=OK;
 int waitpumpack[5];
 int waitpumpctrlack[5];
 int waitsteamack;
 int waitlevelack;
 int waitphysunitsready;

 int mode=mode_init;

 /* These channels are used within the Control Program to store */
 /* states of the pumps */
 chan unusedpumps = [4] of { int };
 chan usedpumps = [4] of { int };
 chan defectpumps = [4] of { int };
 chan defectpumpcontrollers = [4] of { int };
 chan pcsfirst = [4] of { int };
 chan pcssecond = [4] of { int };

 eps[1]=OFF; eps[2]=OFF; eps[3]=OFF; eps[4]=OFF;
 epcs[1]=OFF; epcs[2]=OFF; epcs[3]=OFF; epcs[4]=OFF;

#ifdef MESSAGES
printf(“Control Program is waiting for the Steam Boiler to initialize.\n”);
#endif

 ext_ctrlp?ctrlp_steam_boiler_waiting(0); /* Wait for phys. Units */
                              /* indicating that they are ready. */
#ifdef MESSAGES
printf(“Received Ready Signal from Steam Boiler.\n”);
```

```
#endif

 /* Initially, all Pumps are OK. */
 pumpstate[1]=OK; pumpstate[2]=OK;
 pumpstate[3]=OK; pumpstate[4]=OK;

 /* Give the Pumps free for usage: */
 unusedpumps!1;
 unusedpumps!2;
 unusedpumps!3;
 unusedpumps!4;

Main_Loop:
 do
 :: ext_physunits!physunits_mode(mode) ->

/* Control Program Phase 1: Reception of Messages coming from */
/* the Physical Units */

/* As stated in Section 3 of the Problem Specification, all */
/* Messages should be received/emitted simultaneously */

    lastlevel=level ->
    laststeam=steam ->
    atomic
    {
    ext_ctrlp_pumpstate?ps[1],ps[2],ps[3],ps[4] ->
    ext_ctrlp_pumpctrlstate?pcs[1],pcs[2],pcs[3],pcs[4] ->
    ext_ctrlp?ctrlp_level(level) ->
    ext_ctrlp?ctrlp_steam(steam) ->
    } ->

#ifdef MESSAGES
printf("Mode %d. Boiler: Level %d, Steam %d. Pumps: %d %d %d %d Controllers: %d %d
%d %d\n",mode,level,steam,
ps[1],ps[2],ps[3],ps[4],pcs[1],pcs[2],pcs[3],pcs[4]) ->
#endif

    do
    :: ext_ctrlp?ctrlp_stop(0) -> rec_stop=1
    :: ext_ctrlp?ctrlp_pump_repaired(nr) ->
       if
       :: (nr==1) -> if
                     :: (pumpstate[1]==DEFECT) ->
                        defectpumps??1 -> pumpstate[1]=OK ->
                        unusedpumps!1
                      :: else -> goto Transmissionfailure
                     fi
       :: (nr==2) -> if
                     :: (pumpstate[2]==DEFECT) ->
                        defectpumps??2 -> pumpstate[2]=OK ->
                        unusedpumps!2
                      :: else -> goto Transmissionfailure
                     fi
          :: (nr==3) -> if
                     :: (pumpstate[3]==DEFECT) ->
                        defectpumps??3 -> pumpstate[3]=OK ->
                        unusedpumps!3
                      :: else -> goto Transmissionfailure
                     fi
          :: (nr==4) -> if
                     :: (pumpstate[4]==DEFECT) ->
                        defectpumps??4 -> pumpstate[4]=OK ->
                        unusedpumps!4
                      :: else -> goto Transmissionfailure
                     fi
       fi
       :: ext_ctrlp?ctrlp_pump_control_repaired(nr) ->
           if
          :: (nr==1) -> defectpumpcontrollers??1
          :: (nr==2) -> defectpumpcontrollers??2
          :: (nr==3) -> defectpumpcontrollers??3
          :: (nr==4) -> defectpumpcontrollers??4
          fi
       :: ext_ctrlp?ctrlp_level_repaired(0) ->
          if
          :: (waterlevelunit!=OK) -> waterlevelunit=OK
           :: else -> goto Transmissionfailure
          fi
       :: ext_ctrlp?ctrlp_steam_repaired(0) ->
          if
          :: (steamlevelunit!=OK) -> steamlevelunit=OK
           :: else -> goto Transmissionfailure
          fi

/* Make sure that received Acknowledgments match with sent */
/* Failure Msgs: */
       :: ext_ctrlp?ctrlp_pump_failure_acknowledgment(nr) ->
          if
          :: (waitpumpack[nr]!=ON) -> goto Transmissionfailure
           :: else -> waitpumpack[nr]=OFF
          fi
       :: ext_ctrlp?ctrlp_pump_control_failure_acknowledgment(nr) ->
          if
          :: (waitpumpctrlack[nr]!=ON) -> goto Transmissionfailure
           :: else -> waitpumpctrlack[nr]=OFF
          fi
       :: ext_ctrlp?ctrlp_level_failure_acknowledgment(0) ->
          if
          :: (waitlevelack!=ON) -> goto Transmissionfailure
           :: else -> waitlevelack=OFF
          fi
```

```
    :: ext_ctrlp?ctrlp_steam_outcome_failure_acknowledgment(0) ->
       if
       :: (waitsteamack!=ON) -> goto Transmissionfailure
        :: else -> waitsteamack=OFF
       fi
/* Physical_Units_Ready Messages are only answered */
/* in Initialization Mode: */
    :: ext_ctrlp?ctrlp_physical_units_ready(0) ->
       if
       :: ((waitphysunitsready==1)&&(mode==mode_init))
          -> rec_physunits_ready=1
        :: else -> skip /* This is not necessarily a TX Failure */
       fi
    :: ext_ctrlp?ctrlp_end_of_transmission(0) -> break;
    od;

/* Control Program Phase 2: Analysis of received Information */
/* and Phase 3: Transmission of messages to the Physical Units */
 atomic
 {
 if
 :: (rec_stop==1) -> rec_stop=0 -> stopcounter++ ->
    if
     :: (stopcounter==3) -> goto Emergency_Stop_Mode
     :: else -> skip
    fi
 :: else -> skip
 fi
 } ->

/* First check the Pumps: */
/* in eps[nr] we stored the “expected state” for the next cycle. */
/* by checking this state against the actual pump state, we can */
/* detect pumps that change their state spontaneously as well as */
/* pumps that do not indicate having effectively being started */
/* or stopped */
 if
 :: (ps[1]!=eps[1]) ->
    if
    :: usedpumps??1
     :: unusedpumps??1
     :: defectpumps??1
    fi
    -> atomic
       {
       defectpumps!1
       -> pumpstate[1]=DEFECT -> waitpumpack[1]=ON
       -> eps[1]=ps[1]
       -> epcs[1]=pcs[1]
#ifdef MESSAGES
-> printf(“Defect detected: Pump 1 not in expected state.\n”)
```

```
#endif
         }
 :: else -> skip
 fi ->
 if
 :: (ps[2]!=eps[2]) ->
    if
    :: usedpumps??2
     :: unusedpumps??2
     :: defectpumps??2
    fi
    -> atomic
       {
       defectpumps!2
       -> pumpstate[2]=DEFECT -> waitpumpack[2]=ON
       -> eps[2]=ps[2]
       -> epcs[2]=pcs[2]
#ifdef MESSAGES
-> printf("Defect detected: Pump 2 not in expected state.\n")
#endif
       }
 :: else -> skip
 fi ->
 if
 :: (ps[3]!=eps[3]) ->
    if
    :: usedpumps??3
     :: unusedpumps??3
     :: defectpumps??3
    fi
    -> atomic
       {
       defectpumps!3
       -> pumpstate[3]=DEFECT -> waitpumpack[3]=ON
       -> eps[3]=ps[3]
       -> epcs[3]=pcs[3]
#ifdef MESSAGES
-> printf("Defect detected: Pump 3 not in expected state.\n")
#endif
       }
 :: else -> skip
 fi ->
 if
 :: (ps[4]!=eps[4]) ->
    if
    :: usedpumps??4
     :: unusedpumps??4
     :: defectpumps??4
    fi
    -> atomic
       {
```

```
       defectpumps!4
       -> pumpstate[4]=DEFECT -> waitpumpack[4]=ON
       -> eps[4]=ps[4]
       -> epcs[4]=pcs[4]
#ifdef MESSAGES
-> printf(“Defect detected: Pump 4 not in expected state.\n”)
#endif
       }
    :: else -> skip
 fi ->

/* Next we check the Pump Controllers exactly in the same way */
 if
 :: (pcs[1]!=epcs[1]) ->
    defectpumpcontrollers!1
    -> atomic
       {
       waitpumpctrlack[1]=ON ->
#ifdef MESSAGES
printf(“Defect detected: Pumpcontroller 1 changed State spontaneously.\n”,nr) ->
#endif
       epcs[1]=pcs[1]
       }
 :: else -> skip
 fi ->
 if
 :: (pcs[2]!=epcs[2]) ->
    defectpumpcontrollers!2
     -> atomic
       {
       waitpumpctrlack[2]=ON ->
#ifdef MESSAGES
printf(“Defect detected: Pumpcontroller 2 changed State spontaneously.\n”) ->
#endif
       epcs[2]=pcs[2]
        }
 :: else -> skip
 fi ->
 if
 :: (pcs[3]!=epcs[3]) ->
    defectpumpcontrollers!3
    -> atomic
       {
       waitpumpctrlack[3]=ON ->
#ifdef MESSAGES
printf(“Defect detected: Pumpcontroller 3 changed State spontaneously.\n”) ->
#endif
       epcs[3]=pcs[3]
        }
 :: else -> skip
 fi ->
 if
 :: (pcs[4]!=epcs[4]) ->
    defectpumpcontrollers!4
    -> atomic
       {
       waitpumpctrlack[4]=ON ->
#ifdef MESSAGES
printf("Defect detected: Pumpcontroller 4 changed State spontaneously.\n") ->
#endif
       epcs[4]=pcs[4]
        }
 :: else -> skip
 fi ->

/* Now we should check if the Controller indicates during the */
/* second Cycle after a start or stop message, that the water */
/* is flowing or not flowing despite of the fact that the */
/* program knows from elsewhere that the pump is working correctly*/

 do
 :: (len(pcssecond)!=0) -> pcssecond?nr(state) ->
  if
  :: ps[nr]!=state ->
    if
    :: pumpstate[nr]==DEFECT -> skip /* Make sure Pump is OK */
    :: pumpstate[nr]==OK ->
       defectpumpcontrollers!nr ->
       waitpumpctrlack[nr]=ON ->
#ifdef MESSAGES
printf("Defect detected: Pumpcontroller %d (in 2nd Cycle after switching)\n",nr);
#endif
    fi
     :: else -> skip
  fi
 :: else -> break;
 od;

/* Move data from pcsfirst to pcssecond: */
 atomic
 {
 do
 :: (len(pcsfirst)!=0) -> pcsfirst?nr(state) -> pcssecond!nr(state)
 :: else -> break
 od
 };

/* Next we check the Water Level Measuring Unit by checking if */
/* it indicates either a Value that is out of Range or if it */
/* indicates a Value that is incompatible with the dynamics of */
/* the system. */
/* This must not be done in init-Mode, since the opening of the */
/* valve would result in a difference that is to large... */
 atomic
 {
 if
 :: ( (mode!=mode_init) &&
      ( (level>C) ||
      (level<0) ||
      ( (level-lastlevel)>(5*4*P) ||
      (lastlevel-level)>(5*4*P+W) ) ) )
     -> waterlevelunit=DEFECT
     -> waitlevelack=ON
#ifdef MESSAGES
 -> printf("Defect: Water Level Measuring Unit, last=%d,
now=%d\n",lastlevel,level)
#endif
 :: else -> skip
 fi
 };

/* Now we check the Steam Level Measuring Unit in the same way. */
 atomic
 {
 if
 :: ((steam>W) ||
     (steam<0) ||
     ((steam-laststeam)>2*5) ||
     ((laststeam-steam)>2*5))
    -> steamlevelunit=DEFECT
    -> waitsteamack=ON
 :: else -> skip
 fi
 };

/* Send out Failure Messages ----------------------------------*/
/* (only in degraded and rescue mode) ----------------------- */
 if
 :: ((mode==mode_degraded) || (mode==mode_rescue)) ->
 if
 :: (waitlevelack==ON)
 -> ext_physunits!physunits_level_failure_detection(0)
 :: else -> skip
 fi;
 if
 :: (waitsteamack==ON)
 -> ext_physunits!physunits_steam_failure_detection(0)
 :: else -> skip
 fi;
 if
 :: (waitpumpack[1]==ON)
 -> ext_physunits!physunits_pump_failure_detection(1)
 :: else ->skip
 fi;
 if
 :: (waitpumpack[2]==ON)
 -> ext_physunits!physunits_pump_failure_detection(2)
 ::else -> skip
 fi;
 if
 :: (waitpumpack[3]==ON)
 -> ext_physunits!physunits_pump_failure_detection(3)
 :: else ->skip
 fi;
 if
 :: (waitpumpack[4]==ON)
 -> ext_physunits!physunits_pump_failure_detection(4)
 ::else -> skip
 fi;
 if
 :: (waitpumpctrlack[1]==ON)
 -> ext_physunits!physunits_pump_control_failure_detection(1)
 :: else ->skip
 fi;
 if
 :: (waitpumpctrlack[2]==ON)
 -> ext_physunits!physunits_pump_control_failure_detection(2)
 ::else -> skip
 fi;
 if
 :: (waitpumpctrlack[3]==ON)
 -> ext_physunits!physunits_pump_control_failure_detection(3)
 :: else ->skip
 fi;
 if
 :: (waitpumpctrlack[4]==ON)
 -> ext_physunits!physunits_pump_control_failure_detection(4)
 ::else -> skip
 fi;
 :: else -> skip
 fi;
/* ----- End of Failure Detection Phase ----------------------- */

/* Reactions to Water Level: ---------------------------------- */
 if
 :: ((mode==mode_normal) ||
     (mode==mode_rescue) ||
     (mode==mode_degraded) )
 -> if
    :: ((level>N2) && (len(usedpumps)!=0)) -> usedpumps?nr
        -> ext_physunits!physunits_close_pump(nr)
#ifdef MESSAGES
 -> printf(“Sending CLOSE Command to Pump %d\n”,nr)
```

```
#endif
        -> unusedpumps!nr
       -> eps[nr]=OFF -> epcs[nr]=OFF
 :: ((level<N1) && (len(unusedpumps)!=0)) -> unusedpumps?nr
        -> usedpumps!nr
        -> ext_physunits!physunits_open_pump(nr)
#ifdef MESSAGES
       -> printf(“Sending OPEN Command to Pump %d\n”,nr)
#endif
        -> eps[nr]=ON -> epcs[nr]=ON
/* If the water level is risking to reach one of the limit values */
/* M1 or M2 the Program goes into Emergency_Stop_Mode. This risk */
/* is evaluated on the basis of a maximal behaviour of the */
/* physical units: */
 :: (level>=M2-20*P) -> goto Emergency_Stop_Mode
 :: (level<=M1+20*P) -> goto Emergency_Stop_Mode
 :: ( (level<M2-20*P)&&
      (level>M1+20*P)&&
      ( (level<=N2)||(len(usedpumps)==0) )&&
      ( (level>=N1)||(len(usedpumps)==0) ) ) -> skip /* else */
 fi

/* In initialization Mode, only pump_on and open_valve commands */
/* are possible */
 :: (mode==mode_init)
    -> if
       :: ((level<N1) && (len(unusedpumps)!=0)) -> unusedpumps?nr
           -> usedpumps!nr
           -> ext_physunits!physunits_open_pump(nr)
#ifdef MESSAGES
 -> printf(“Sending OPEN Command to Pump %d\n”,nr)
#endif
           -> eps[nr]=ON -> epcs[nr]=ON
        :: ((level>N2)) -> ext_physunits!physunits_valve(0)
#ifdef MESSAGES
 -> printf(“Sending OPEN VALVE Command.\n”)
#endif
       :: ( (level>=N1)&&
            (level<=N2) ) ->
        if
        :: (rec_physunits_ready!=1) ->
          ext_physunits!physunits_program_ready(0) ->
           waitphysunitsready=1
         :: (rec_physunits_ready==1) -> skip
       fi
 :: else -> skip
 fi
fi;

/* Eventually Change the Control Programs Operating Mode -------- */
 atomic
 {
 if
 :: (mode==mode_normal)
     -> if
        :: (waterlevelunit==DEFECT) -> mode=mode_rescue
        :: ( (waterlevelunit==OK) &&
             ( (steamlevelunit==DEFECT)||
             (len(defectpumpcontrollers)!=0)||
             (len(defectpumps)!=0) ) ) -> mode=mode_degraded
        :: ( (waterlevelunit!=DEFECT) &&
             ( (steamlevelunit!=DEFECT)&&
             (len(defectpumpcontrollers)==0)&&
             (len(defectpumps)==0) ) ) -> skip /* else */
        fi
 :: (mode==mode_init)
     -> if
          :: ( (level>=N1)&&(level<=N2)&&(rec_physunits_ready==1)) ->
            rec_physunits_ready=0 ->
            if
            :: ((len(defectpumps)==0) &&
                (len(defectpumpcontrollers)==0) &&
                (steamlevelunit==OK) &&
                (waterlevelunit==OK)) -> mode=mode_normal
            :: else -> mode=mode_degraded
            fi
          :: else -> skip
         fi
 :: (mode==mode_rescue)
    -> if
        :: ( (waterlevelunit==OK) &&
             ( (steamlevelunit==DEFECT)||
             (len(defectpumpcontrollers)!=0)||
             (len(defectpumps)!=0) ) ) -> mode=mode_degraded
        :: ( (waterlevelunit==OK) &&
             (steamlevelunit!=DEFECT) &&
             (len(defectpumpcontrollers)==0) &&
             (len(defectpumps)==0) ) -> mode=mode_normal
        :: else -> skip
        fi
 :: (mode==mode_degraded)
    -> if
       :: (waterlevelunit!=OK) -> mode=mode_rescue
       :: ( (waterlevelunit==OK)&&
            (steamlevelunit==OK)&&
            (len(defectpumps)==0)&&
            (len(defectpumpcontrollers)==0) ) -> mode=mode_normal
       :: else -> skip
      fi
 fi
 };
od;
```

```
Transmissionfailure:
 skip;
#ifdef MESSAGES
printf("Transmission Failure between Physical Units and Control Program\n");
printf("-> Going into Emergency Stop Mode.\n");
#endif

Emergency_Stop_Mode:
ext_physunits!physunits_mode(mode_emergency);
#ifdef MESSAGES
printf("Emergency Stop!\n");
#endif

end:
skip;
#ifdef MESSAGES
printf("End.\n");
#endif
}
```

5.1.8 Source Code for the Physical Units

The physical units section consists of the process prototypes for the various physical units plus the "`Physical_Units`" proctype which is responsible for the communication with the control program and coordinates all physical units. Between the physical units there are some shared (globally defined) variables for the states of water level meter, steam meter, pumps and pump controllers. There are also some local communication channels between the "`Physical_Units`" proctype and the other proctypes.

```
/* Communication Channel Declarations -----------------------*/
chan valve = [1] of { int }; /* Messages to the Valve */
chan pumps = [10] of { int, int };
chan boiler = [1] of { int, int, int};

/* Message Identifiers ---------------------------------------- */
/* Messages between the physunits-Proctype and the "real" */
/* Components of the Physical Units Segment, that is Pumps, */
/* Valve, Boiler, ... */
#define valve_activate 100
#define pumps_open 200
#define pumps_close 201
```

```
#define pumps_ack 202
#define pumps_terminate 203
#define pumps_getstate 204
#define pumps_flow 205
#define pumps_askpumpswitches 206
#define pumps_switchstates 207
#define pumps_askpumpctrlsenses 208
#define pumps_pumpctrlsenses 209
#define boiler_cycle 500
#define boiler_status 501
#define boiler_terminate 502


/* Global Variables for the Physical Units -------------------- */
/* (these are not to be used by the Control Program) ---------- */

/* Variables to mark defective Units */
int levelmeterstate;
int steammeterstate;
int pumpctrl_state[5];

int pump_switch[5];
int pumpctrl_sense[5];

int pflow;

/* ----------------- PHYSICAL UNIT SECTION -------------------- */
proctype Pumps()
{
 int pumpid;

 atomic
 {
 pumpctrl_state[1]=OK; pumpctrl_state[2]=OK;
 pumpctrl_state[3]=OK; pumpctrl_state[4]=OK;

 pump_switch[1]=OFF; pump_switch[2]=OFF;
 pump_switch[3]=OFF; pump_switch[4]=OFF;

 pumpctrl_sense[1]=OFF; pumpctrl_sense[2]=OFF;
 pumpctrl_sense[3]=OFF; pumpctrl_sense[4]=OFF;
 };

end:
 do
 :: pumps?pumps_open(pumpid)
    -> if
        :: pump_switch[pumpid]==OFF ->
           atomic
          {
           pump_switch[pumpid]=ON ->
          pflow=pflow+P ->
#ifdef MESSAGES
printf("Pump %d ON.\n",pumpid) ->
#endif
        if
        :: (pumpctrl_state[pumpid]==OK) ->
          pumpctrl_sense[pumpid]=ON
        :: (pumpctrl_state[pumpid]!=OK) ->
          skip
        fi
       }
    :: else -> skip
    fi -> pumps!pumps_ack(pumpid)

 :: pumps?pumps_close(pumpid)
    -> if
       :: pump_switch[pumpid]==ON ->
          atomic
          {
           pump_switch[pumpid]=OFF ->
          pflow=pflow-P ->
#ifdef MESSAGES
        printf("Pump %d OFF.\n",pumpid) ->
#endif
          if
           :: (pumpctrl_state[pumpid]==OK) ->
              pumpctrl_sense[pumpid]=OFF
           :: (pumpctrl_state[pumpid]!=OK) ->
              skip
          fi
           }
        :: else -> skip
    fi -> pumps!pumps_ack(pumpid)

 :: pumps?pumps_terminate(0) -> goto End_Pumps
od;
End_Pumps:
skip;
#ifdef MESSAGES
printf("Pump process terminated.\n");
#endif
}


/* The Boiler has no communication Channels except the Timer /*
/* Channel, which is used to keep it in Sync with the Control */
/* Program, whose Cycle Time is 5 Seconds */
proctype Boiler()
{
 int q,v,pf,heating;
 do
 :: boiler?boiler_cycle(pf,heating) ->
     atomic
    {
    q=q-(5*v)+(5*pf) -> /* in:p, out:v */
#ifdef MESSAGES
printf("Boiler holds %d Liters of Water and produces %d Liters of Steam/
sec.\n",q,v) ->
#endif
    if
    :: (heating==ON) -> v = (q/100) /* v is integer... */
     :: (heating==OFF) -> v = 0
    fi
     }
    -> boiler!boiler_status(q,v)
 :: boiler?boiler_terminate(0,0) -> goto End_Boiler;
 :: valve?valve_activate -> q=0; v=0
 od;

End_Boiler:
skip;
#ifdef MESSAGES
printf("Boiler Process terminated.\n");
#endif
}

/* The "Physical Units" have to be seen as a logical Group of */
/* all physical Units in the Steam Boiler. They are connected */
/* via a communication Networkto the Control Program. The */
/* Control Program can only communicate with the */
/* Physical_Units() Proctype, which coordinates all Units */
proctype Physical_Units()
{
 int nr;
 int ctrlpmode;
 int rec_program_ready = 0;
 int rec_level_failure_detection = 0;
 int rec_steam_failure_detection = 0;
 int rec_pump_repaired_acknowledgment = 0;
 int rec_pump_control_repaired_acknowledgment = 0;
 int rec_level_repaired_acknowledgment = 0;
 int rec_steam_repaired_acknowledgment = 0;
 int blevel,bsteam;

 chan rec_pump_failures = [4] of { int };
 chan rec_pump_control_failures = [4] of { int };

 /* Start up all Physical Units */
 atomic
 {
 run Pumps();
 run Boiler();
 }
 /* Now notify the Control Program that we are ready to go... */
 ext_ctrlp!ctrlp_steam_boiler_waiting(0);

 ext_physunits?physunits_mode(ctrlpmode) ->

physunits_main:
 do
 :: if
 :: (ctrlpmode==mode_init) ->
     boiler!boiler_cycle(pflow,OFF) /* No heating in Init Mode */
     :: (ctrlpmode!=mode_init) ->
 boiler!boiler_cycle(pflow,ON)
 fi ->
 boiler?boiler_status(blevel,bsteam) ->
 /* A Mode Message must always be answered with Messages that */
 /* must be present during each transmission: */
 atomic
 {
     ext_ctrlp_pumpstate!pump_switch[1],pump_switch[2],
     pump_switch[3],pump_switch[4] ->
     ext_ctrlp_pumpctrlstate!pumpctrl_sense[1],pumpctrl_sense[2],
     pumpctrl_sense[3],pumpctrl_sense[4] ->
    ext_ctrlp!ctrlp_level(blevel) ->
    ext_ctrlp!ctrlp_steam(bsteam)
     } ->
    if
    :: (ctrlpmode==mode_emergency) -> goto End_Physunits
     :: else -> skip
 fi;

   /* Afterwards we transmit optional Messages: */
 do
     :: (rec_program_ready==1)
        -> ext_ctrlp!ctrlp_physical_units_ready(0)
        -> rec_program_ready=0
     :: (len(rec_pump_failures)!=0)
        -> rec_pump_failures?nr
         -> ext_ctrlp!ctrlp_pump_failure_acknowledgment(nr)
     :: (len(rec_pump_control_failures)!=0)
        -> rec_pump_control_failures?nr
        -> ext_ctrlp!ctrlp_pump_control_failure_acknowledgment(nr)
     :: (rec_level_failure_detection!=0)
        -> ext_ctrlp!ctrlp_level_failure_acknowledgment(0)
        -> rec_level_failure_detection=0
     :: (rec_steam_failure_detection!=0) ->
        ext_ctrlp!ctrlp_steam_outcome_failure_acknowledgment(0) ->
        rec_steam_failure_detection=0
     :: else -> break
 od;
```

```
    /* And finally we mark the End of our Transmission */
    ext_ctrlp!ctrlp_end_of_transmission(0);

 do
     /* All following are answered in the _next_ Cycle... */
 :: ext_physunits?physunits_program_ready(0)
    -> rec_program_ready=1;
 :: ext_physunits?physunits_open_pump(nr) -> pumps!pumps_open(nr)
    -> pumps?pumps_ack(nr)
 :: ext_physunits?physunits_close_pump(nr) -> pumps!pumps_close(nr)
    -> pumps?pumps_ack(nr)
 :: ext_physunits?physunits_valve(0) -> valve!valve_activate

 :: ext_physunits?physunits_pump_failure_detection(nr)
    -> rec_pump_failures!nr
 :: ext_physunits?physunits_pump_control_failure_detection(nr)
     -> rec_pump_control_failures!nr
 :: ext_physunits?physunits_level_failure_detection(0)
    -> rec_level_failure_detection=1
 :: ext_physunits?physunits_steam_failure_detection(0)
     -> rec_steam_failure_detection=1
 :: ext_physunits?physunits_pump_repaired_acknowledgment(nr)
    -> rec_pump_repaired_acknowledgment=1
:: ext_physunits?physunits_pump_control_repaired_acknowledgment(nr)
    -> rec_pump_control_repaired_acknowledgment=nr
 :: ext_physunits?physunits_level_repaired_acknowledgment(0)
     -> rec_level_repaired_acknowledgment=1
 :: ext_physunits?physunits_steam_repaired_acknowledgment(0)
    -> rec_steam_repaired_acknowledgment=1
 :: ext_physunits?physunits_mode(ctrlpmode) -> goto physunits_main
od;
od;
End_Physunits:
#ifdef MESSAGES
printf(“Physunits received Emergency Stop Message.\n”);
#endif
/* Once the Program has reached Emergency Stop MOde, the physical */
/* Environment is then responsible to take appropriate actions: */
valve!valve_activate ->
pumps!pumps_close(1) -> pumps?pumps_ack(1) ->
pumps!pumps_close(2) -> pumps?pumps_ack(2) ->
pumps!pumps_close(3) -> pumps?pumps_ack(3) ->
pumps!pumps_close(4) -> pumps?pumps_ack(4) ->
pumps!pumps_terminate(0) ->
boiler!boiler_terminate(0,0) ->
#ifdef MESSAGES
printf(“Physunits stopped. (Emergency Stop).\n”);
#endif
}
```

5.1.9 Refining the Specification

As mentioned earlier, this PROMELA specification is not yet covering the full problem. All devices that are modelised in the specification are always working correct. However we already have failure recognition routines in the control program, so the only extensions that are necessary are within the physical units.

Unfortunately, the state space of the validation model already is quite large so that it already was impossible to do a exhaustive validation on a machine with 64 MB memory. By adding device failures to the specification, the state space will even get larger and a validation of the problem will get more difficult.

When specifying the steam boiler with PROMELA, we made quite some experiences on the applicability of the language to this kind of problem. The first experience we made is the importance of the structurisation of the problem. It seems advisable to structurise the source code in a way that resembles the real physical structures and to start by writing proctypes for each entity that exists in reality. Doing so the informal specification can be partly “translated” into the formal specification, which makes it much easier to get to a working model. Afterwards one can try to simplify the formal specification by eliminating process protoypes that do almost nothing or by integrating multiple actions in atomic sequences and so on.

5.2 An external Client for User-Communication

5.2.1 Purpose of a User-Client

As there is no possibility within PROMELA to read data from the user, we thought that it might be useful to create a client that deals with this task. From PROMELA, the user data can then be read via an external channel. Additionally, we implemented a channel that can be used to output data to the user.

```
proctype comm()
{
byte y=0;

do
 :: set_timer!_pid(10);
   if
   :: ext_user?y; del_timer!_pid;
      printf(“Received: %d\n”,y);
   :: timer?y; ext_printf!1;
      printf(“Timed out without User Data...\n”,y);
   fi
od
}
```

Figure 41: How the User-Client can be accessed from within a PROMELA Specification

For the communication with the client, the two channels “`ext_user`” (for reading “`byte`” type data) and “`ext_printf`” have to be used, as can be seen in the example depicted in Figure 41.

Of course, the use of this client brings some problems for validation and simulation with it. For validation it would be necessary to create a PROMELA proctype that simulates the behaviour of the user. Nevertheless, we think that this is external client is a good example for how an external client can be written in C. The following Listings show the Source Code for the client.

5.2.2 The Source Code for the Client

The client consists of two files, “`user.c`”, which contains the “`main()`” routines - the heart of the client and “`userlink.h`”, which is mainly a copy of the file “`comcli.h`” that the modified SPIN generates with its clients.

The following is the listing for “`user.c`”:

```
struct fd_set inputfdmask; /* Mask for select() on sockets... */
struct fd_set selectfdmask;
int nrfds=0;

char *buffer,*buffer1;
int buffersize=1024;

#include <types.h>
#include <time.h>
#include “userlink.h” /* This is about the same as the comcli.h */
                       /* (Routines for clients...) */

void
main(int argc, char *argv[])
{
 int t;

 int bc=0;
 long bb;
 char b;

 char msgtype;
 char channame[16] = “ext_user\0”;
 int qlen,value;


 buffer=(char *)malloc(buffersize);
 buffer1=(char *)malloc(buffersize);
 (void) memset(buffer, 0, buffersize);
 (void) memset(buffer1,0, buffersize);

 FD_ZERO(&inputfdmask);    /* Reset (initialise) the fd-mask */
 FD_SET(0,&inputfdmask);   /* Add stdin */
 nrfds++;

 /* Connect to the Server Process via Socket */
 connect_server();
 for(;;)
 {
 (void) memcpy(&selectfdmask, &inputfdmask, sizeof(fd_set));
#ifdef DEBUG
 printf("("); fflush(0);
#endif DEBUG
 t=select(32,&selectfdmask, NULL, NULL ,NULL);
#ifdef DEBUG
 printf(")"); fflush(0);
#endif DEBUG

 if(t==-1)
    {
     perror("select ");
     exit(1);
    }
 else if(t==0)
    {
     printf("Timeout.\n");
    }

 /* Something is available for input - can be either a message */
 /* from the server or user input from stdin. */

 if(FD_ISSET(0, &selectfdmask)) /* Data on stdin... */
    {
     t=read(0,&b,1);
#ifdef DEBUG
     printf("%c",b);
#endif
     if((b!=13)&&(b!=10)&&(b>=48)&&(b<=57))
     /* Add one Character to the chain */
     {
     buffer[bc]=b; bc++;
     }
     else if( ((b==13)||(b==10)) &&(bc!=0))
     {
     buffer[bc]=0;
     printf("Send: %s\n",buffer);

     (void) sprintf(buffer1, "!%s %d %s\n", channame, 1, buffer);
     do_send(extsock, buffer1, strlen(buffer1));

     (void) memset(buffer,0,bc+1);
     bc=0;
     }
    }
 else
    {
     /* Reception of a Message... */
     do_recv(extsock, buffer, buffersize);
#ifdef DEBUG
     printf(“recv< %s\n”,buffer);
#endif
     msgtype=’ ‘;
     sscanf(buffer,”%c%s %d”,&msgtype,channame,&qlen);

     /* Now handle the Received Message */
     if(msgtype==’@’)
     {
     /* We were notified that there was Data... */
     if((strcmp(channame,”ext_printf”)==0) && (qlen!=0))
        {
         (void) memset(buffer1,0,buffersize);

         /* Ask the Server to send the Data and delete it in Queue */
         sprintf(buffer1,”/ext_printf 1 0”);
         do_send(extsock,buffer1,buffersize);
        }
     }
     else if(msgtype==’!’)
     {
     if(strcmp(channame,”ext_printf”)==0)
        {
         sscanf(buffer,”%c%s %d %d”,&msgtype,channame,&qlen,&value);
         printf(“Message received: %d\n”,value);
        }
     }
    }
 }
}
```

6 Conclusions and Outlook

In producing 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. 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.

Appendix A: Message Format

Between the server and the clients, a very simple, ASCII based message format is being used. Fig-

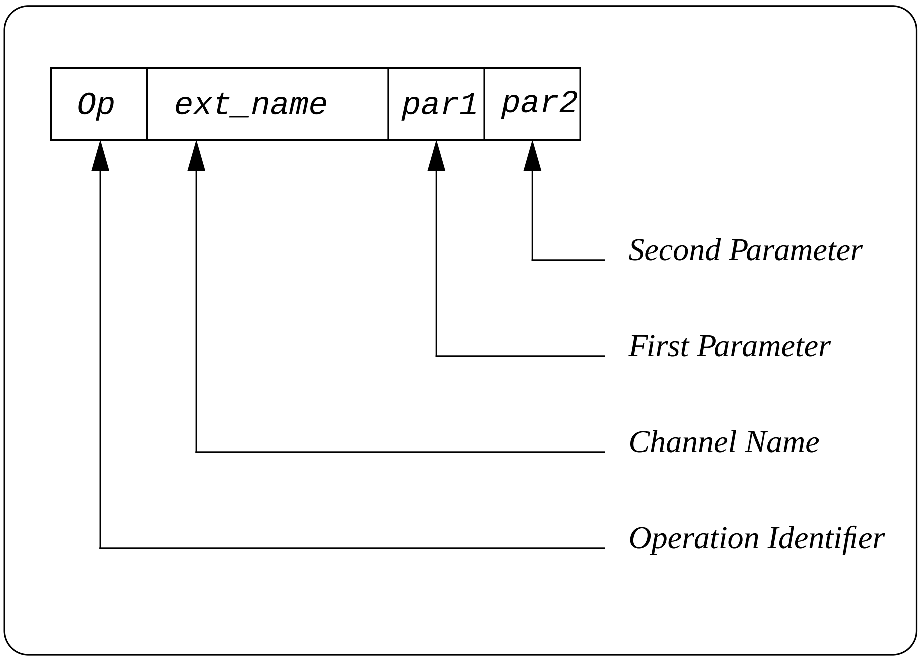


Figure 42: Message Format

ure 42 shows the general format of the messages. The first field is the “Operation Identifier”, which determines the type of the message. Afterwards, separated by a white space, the channel name (up to 16 characters long) is transmitted, followed by up to two parameters. This channel name is derived from the name in the PROMELA specification, therefore the names in the specification should not be longer then 16 characters. The message is terminated with a carriage return. This simple format has the advantage that it is very suitable for debugging purposes, because the transmitted messages are coded in ASCII and therefore are directly readable by humans. The disadvantage is of course that bandwidth is wasted.

Table 2 on page 90 shows the possible message types that have been implemented in the code for communications server and client (“`comserv.h`” and “`comcli.h`”). In order to write own clients, it is probably the easiest way to copy the file "`comcli.h`" from a client that was generated with SPIN and reuse the routines that are defined in this file. It has to be taken care that the client really receives all messages of type "@" . Those are sent out by the server each time that any of the external channels data structures changes. Even if a client is not interested in this information, it still has to read the messages because otherwise the buffers for communications between client and server might overflow.

TABLE 2. Message Types

Operation Identifier	**From / To**	**Purpose**	**Parameters**
@	Server ➔ All clients	Notification about changes in a channels contents	*`Par1`*=Current number of filled FIFO queue places
!	Server ➔ Client	Transmission of contents of a channels FIFO queue	*`Par1`*=Number of fields in transmissions *`Par2`*=Value
/	Client ➔ Server	Read a byte from a queue and delete nflds	*`Par1`*=Number of fields in transmissions *`Par2`*=Number of field to read
-	Client ➔ Server	Read a byte from a queue without deletion ("peek")	*`Par1`*=Number of fields in transmissions *`Par2`*=Number of field to read

Appendix B: Listings

B.1: The Scheduler

```
/* $Id: com.src.c,v 1.13 1996/02/21 16:46:59 loeffler Exp loeffler $ */

/* Runtime -*- C -*- */
/* S. Loeffler 1995, 96 */

static char vcid[]=”$Id: com.src.c,v 1.13 1996/02/21 16:46:59 loeffler Exp loef-
fler $”;

#define RUNTIME /* Disables Parts of the Analyzer-Code */
#define MAXPARS 100 /* Max Buffersize for ext. Send/Receive */

#define MASK(f) (1 << (f)) /* For computing select() masks */
#define TRUE 1
#define FALSE 0


/* -- Global Variables --------------------------------------*/
char *buffer;       /* Pointers to the Buffers for ext Comm. */
char *buffer1;
int buffersize;                      /* Size of the Buffers */

long next_tim=0;  /* Time (System Clock) for next Timer event */
long cur_tim;             /* Current Time (Unix System Clock) */

int nrfds;         /* Number of File Descriptors for Select() */
struct fd_set inputfdmask;    /* fd-mask for Select(), stored */
struct fd_set selectfdmask; /* fd-mask for Select(), argument */
/* ------------------------------------------------------- */

#ifdef SERVER
#include “comserv.h”      /* Procedures for ext. comm. server */
#endif

#ifdef CLIENT
#include “comcli.h” /* or Client */
#endif

#include “comstmng.h” /* Procedures from the Analyzer-Code */

#include <time.h>
```

```
/* ---------------------------------------------------------- */

/* Synchronous Channels (blocking) is done a bit different in */
/* the implementation... */
#if SYNC
int blocked[MAXPROC]; /* Mark proctypes as blocked for sync.op. */
int blockedsender[MAXQ];      /* Proctype ID of Sender Process */
int tr_qid=0;       /* Which synchronous queue is responsible? */
#define IfNotBlocked if(blocked[II] !=0) continue;
#define UnBlock blocked[II]=0;
#else
#define IfNotBlocked /* No synchronous channels -> cannot block */
#define UnBlock      /* No synchronous channels -> no blocking */
#endif

/* Data Structures for Realtime-Timers */
#ifdef HAS_TIMERS
struct timer
{
 int nr; /* Nr. of Timer */
 long val; /* Timer Value (like Unix System Clock) */
 struct timer *nxt;
};
struct timer *timers;
#endif

#ifdef HAS_TIMERS
struct timer
*remove_timer(struct timer *p, int remtr_nr)
{
 struct timer *t, *pred;
 long minval=0;
 int xx;

 pred=NULL; /* Predecessor */

 for(t=p; t!=NULL ; t=t->nxt)
 {
 if (t->nr==remtr_nr)
    {
     if (pred==NULL) /* Deleting the First one */
     {
     if (t->nxt==NULL) /* Deleting the only one */
        {
         p=NULL;
         free(t);
         break;
        }
     else
        {
         p=t->nxt;
        free(t);
        break;
       }
     }
     else
     {
     pred->nxt=t->nxt;
     free(t);
     break;
     }
    }
 pred = t;
 }

#ifdef DEBUG
 printf("Timer %d was deleted from Timer List.\n",remtr_nr);
#endif

 /* Remove the "fired" signal fm recvd list in case it exists */
 if((xx=Q_has(now.timer, 1, remtr_nr, 0, 0)))
 {
 printf("WARNING: Removing the Signal that the Timer already sent.\n");
 qrecv(now.timer,xx-1,0,1);
 }

 /* Find the next one that will fire... */
 for(t=p; t!=NULL; t=t->nxt)
 {
 if((t->val<minval))
    {
     minval=t->val;
    }
 }
 next_tim=minval;

 return(p);
}


struct timer
*new_timer(void)
{
 return (struct timer *)malloc(sizeof(struct timer));
}


struct timer
*add_timer(struct timer *p, int addt_nr, int addt_val)
{
 struct timer *pp, *pq, *t;
 /* First check if that Timer already exists. */
 for(t=p; t!=NULL; t=t->nxt)
 {
 if (t->nr == addt_nr)
    {
     printf("WARNING: Timer %d already exists. Deleting the old one,\n",addt_nr);
     printf(" but you are advised to add an \"del_timer\" in the specifica-
tion\n");
     p=remove_timer(timers, addt_nr);
     break;
    }
 }

 pp = new_timer();
 pp->nxt = 0;
 pp->nr = addt_nr;
 cur_tim = time(0);
 pp->val = cur_tim + (long)addt_val;

 if(p==NULL) /* First one */
 {
 p=pp;
 }
 else
 {
 pq = p;
 while (pq->nxt!=NULL)
    {
     pq=pq->nxt;
    }
 pq->nxt = pp;
 }

 if((addt_val+cur_tim<next_tim)||(next_tim==0))
 {
 next_tim=addt_val+cur_tim;
 }
 return(p);
}


void
handle_timers(void)
{
 struct timer *t;

 /* Now we check the Timer List for waiting Timers... */
 for(t=timers; t!=NULL ; t=t->nxt)
 {
#ifdef DEBUG
 printf("Checking #%d (Val: %Ld) against Utime:%Ld\n",t->nr,t->val,cur_tim);
#endif

 if (t->val<=cur_tim)
    {
#ifdef DEBUG
     printf("Timer Nr %d Value %d fires.\n",t->nr, t->val);
#endif
     timers=remove_timer(timers, t->nr);

     if (q_full(now.timer))
     {
     printf("Fatal Error: \"timer\" Queue Full.\n");
     printf("Cannot send Timeout Message.\n");
     exit(1);
     }
     qsend(now.timer, 0, t->nr, 0);
    }
 }
}


void
check_timers(void) /* This is the Monitor Process. Executed after */
 /* each non-atomic Transition, that is after */
 /* each Proctype-change */
 /* Keep this short, as it takes much CPU Time !*/
{
 int ct_ti_nr, ct_ti_val;
 int ti_nr;


 /* Check for Messages to the Monitor-Process */
 if (q_len(now.set_timer) != 0)
 {
 /* Add Timer */
 ct_ti_nr = (int) qrecv(now.set_timer, 0, 0, 0);
 ct_ti_val = (int) qrecv(now.set_timer, 0, 1, 1);
/* done=1 --> get Data out of the Queue */
#ifdef DEBUG
 printf("Timer %d will be set to %d \n",ct_ti_nr, ct_ti_val);
#endif
 timers=add_timer(timers, ct_ti_nr, ct_ti_val);
 }

 if (q_len(now.del_timer) != 0) /* Reset Timer Message received */
 {
 /* Delete Timer */
 ti_nr = (int) qrecv(now.del_timer, 0, 0, 1);
 timers=remove_timer(timers, ti_nr);
 }
 cur_tim=time(0);
 if((next_tim!=0)&&(next_tim<=cur_tim)) /* There is a Timer that fires */
 {
 handle_timers();
 }
}
#endif /* HAS_TIMERS */

/* ************* Main Runtime ****************************** */

void
runtime(void)
{
 short executedany;
 short executedfirstatomic;

 int i;
 struct timeval s_timeout;

 register Trans *t;
 char ot,m;
 short flag_atom;
 short tt;
 short II;

 short From, To; /* Range of (active) Proctypes being searched */

 FD_ZERO(&inputfdmask); /* Reset the fd-mask */
 FD_SET(0,&inputfdmask);
 FD_SET(1,&inputfdmask);

 stack = (Stack *)emalloc(sizeof(Stack));

 /* a place to point for Pptr of non-running procs: */
 noptr= (uchar *)emalloc(Maxbody * sizeof(char));

 buffersize=20+MAXPARS*13;
 buffer=(char *)malloc(buffersize); /* Buffer for ext. Communications */
 buffer1=(char *)malloc(buffersize);

 run(); /* Initializations of some Arrays etc. */
 /* This is just the HEAD of the */
 /* Analyzers run() Procedure! */

 hinit(); /* Initialize the State Vector */
 /* (allocate Memory) */

 active_procs(); /* calls addproc(0) to add the */
 /* init proctype */

#if SYNC
```

```
 for(II=0; II<=MAXPROC; II++) blocked[II]=0;
#endif

#ifdef SERVER
 initialize_server(); /* Create AF_UNIX Socket on which the */
 /* Server will listen for requests */
#endif

#ifdef CLIENT
 connect_server();
#endif

MainLoop:
 executedany=FALSE;
 From=now._nr_pr-1;
 To=0;

 for (II=From; II>=To; II-=1) /* For all active Proctypes... */
 {
#ifdef HAS_TIMERS
 check_timers();
#endif

#ifdef SERVER
 checksocks();
#endif

 this=pptr(II);
 ot=(uchar) ((P0 *)this)->_t; /* Current Proctype */
 executedfirstatomic=FALSE; /* Flag to mark if we are allowed */
 /* to change the Proctype */

lbl_atom: /* if atomic move, dont leave this proctype */
 tt=(short) ((P0 *)this)->_p; /* Current State */

#if SYNC
 if (blocked[II]!=0) /* synchronous operations may lock the proc */
 {
#ifdef DEBUGSTATE
   printf(“Locked on State %d\n”,tt);
#endif
   goto lbl_nxt;
 }
#endif /* SYNC */


 /* Now that we have selected a proctype */
 /* search it for executable transitions */
#ifdef DEBUGSTATE
 printf(“Current: Proctype %d/[%d..%d] (ID %d) State %d \n”,II,From,To,ot,tt);
#endif
```

```
 /* choose one... */
 for (t=trans[ot][tt]; t; t=t->nxt) /* loop over transitions */
                                   /* for this proctype/state */
 {
#ifdef DEBUGSTATE
   printf(“Select Transition [id %d] tr %d \”%s\” (to go to State %d).\n”,
    t->t_id,t->forw,t->tp,t->st);
#endif

   if (t->atom&2)
    {
#ifdef DEBUGSTATE
    printf(“atom(%d) Atomic Bit is set.\n”,t->atom);
#endif
    flag_atom = 1;
    }
   else
    {
    flag_atom = 0;
    }

   /* Try to execute the Transition */
#include “com.m”
   /* if it is not executable, it does a continue; */
   /* otherwise it jumps to P999 (see below) */

#ifdef DEBUGSTATE
   printf(“It failed.\n”);
#endif

#if SYNC
   if((tr_qid)!=0)  /* Tried to access a synchronous channel */
    {
    blocked[II]=tt; /* retry until this is successful */
    }
#endif

 } /* end for t=... loop */

 /* there is no t->nxt... */

#ifdef DEBUGSTATE
 printf(“There is no possible move for this state.\n”);
#endif

 /* TODO: Check if handling of atomic Mode is OK. */
 if((flag_atom==1)&&(executedfirstatomic))
 {
   goto lbl_atom;
 }
 else
 {
   goto lbl_nxt;
 }


 /* com.m jumps here if the move is successful... */
P999:
#ifdef DEBUGSTATE
 printf(“It suceeded.\n”);
#endif
 executedany=TRUE;
 if(flag_atom)
 {
   executedfirstatomic=TRUE;
 }

 if (t->st >0)
 {
   ((P0 *)this)->_p = t->st; /* Set “this”_p to the new State */
 }
 else
 {
#ifdef DEBUGSTATE
   printf(“Terminating this UNIX Process. (End)\n”);
#endif
   exit(0);
 }

#if SYNC
 /* Go to “blocked” state. */
 /* com.m returns tr_qid=0 if asynchronous channel */
 /* was accessed, otherwise it returns the channel identifier */
 /* Blocked on receive: Wait for Data (m=4) */
 /* Blocked on send: Wait for Receive (m=2) */
 if((m==2)&&(tr_qid!=0)) /* This was a synchronous send. */
 {
#ifdef DEBUG
   printf(“Did a synchronous send. -> Queue=%d (Proctype %d
blocks)\n”,tr_qid,II);
#endif
 blocked[II]=tt;
   blockedsender[tr_qid]=II;
 }
 else if((m==4)&&(tr_qid!=0)) /* This was a synchronous receive */

 {
#ifdef DEBUG
    printf(“Synchronous receive empties Queue -> unblock. \n”);
#endif
    blocked[II]=0; /* unblock receiver. */
```

```
     /* if queue is extern -> send notification message */
     /* TODO! */

     /* if queue is intern -> set blocked[sender]=0 */
     blocked[blockedsender[tr_qid]]=0;
     blockedsender[tr_qid]=0;
 }
#endif /* SYNC */

 if ((flag_atom==1)&&(executedfirstatomic))
 {
 goto lbl_atom; /* Only if the first atomic Instruction was */
   /* really executed, we have to stay in the same Proctype */
 }

 lbl_nxt:
 } /* next active proctype*/

EndOfMainLoop:
 /* Here is the end of the “MainLoop”. */
 /* -> we have tried all possible transitions. */

 if(!executedany)
 {
 /* If absolutely nothing (in no proctype) was executable, block */

 /* Check if there is a Timer set */
#ifdef HAS_TIMERS
 cur_tim = time(0);
 if(next_tim>cur_tim)
 {
    s_timeout.tv_sec=next_tim-cur_tim;
    s_timeout.tv_usec=0;
 }
 else
 {
    s_timeout.tv_sec=1; /* Just to make sure there is a timeout */
    s_timeout.tv_usec=0;
 }
#else /* if no timers exist, we set the timer limit to one sec... */
 s_timeout.tv_sec=1;
 s_timeout.tv_usec=0;
#endif

#ifdef DEBUG
    printf(“Select (Timeout=%d s) ...”,s_timeout.tv_sec);
#endif

 (void) memcpy(&selectfdmask, &inputfdmask, sizeof(inputfdmask));
 i=select(nrfds+5,&selectfdmask,NULL,NULL,&s_timeout);
 if(i==-1)
 { /* An Error occured in select() */
   printf(“... Error.\n”);
   perror(“select “);
 exit(1);
 }
 else if(i==0)
 {
#ifdef DEBUG
 printf(“... Timeout\n”);
#endif
 }
 else
 {
#ifdef DEBUG
   printf(“... Signal\n”);
#endif
 }

 }
goto MainLoop;
}

/* ----------------------------------------------------------- */


/* For the Runtime System, we have our own wrapup() Function */

void
rt_wrapup(int arg)
{
 (void) signal(SIGINT, SIG_DFL); /* Reset SIGINT to the Default Action */
 printf(“\n\nInterrupt.\nExecutable stopped.\nRuntime: %s\n”,vcid);
 exit(arg);
}

void
main(int argc, char *argv[])
{
/* FILE *fd_e = stderr; */
 signal(SIGINT, rt_wrapup);
 runtime();

 exit(0);
}
```

B.2: Communication Code for Clients

```
/* $Id: comcli.src.c,v 1.3 1996/04/11 09:34:46 loeffler Exp loeffler $ */

/* EXTERNAL COMMUNICATION ROUTINES, CLIENT CODE */

/* This file contains the procedures that are necessary for */
/* external communication between the compiled PROMELA code */
/* and other UNIX processes (which may either be compiled PROMELA */
/* or something different). */
/* The code in this File is for the CLIENT, which does not store */
/* any channel data but has to communicate with the server a lot. */
/* (c) 1995,96 by Siegfried Loeffler */
/* Ecole Nationale Superieure des Telecommunications, Paris */
/* Networks Department / Dept. Reseaux */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* for AF_UNIX */
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>

#ifdef SOLARIS
#include <sys/filio.h> /* for FIONREAD */
#include <stropts.h>
#include <poll.h>
#endif

/* Function Prototypes ------------------------------------- */
extern int unlink();
extern void free();
extern char* malloc();
extern char* emalloc();
/* --------------------------------------------------------- */

#define MSGSIZE 1024

int extsock; /* File Descr ref. to socket */

char extbuffer[MSGSIZE];

extern char *buffer,*buffer1; /* defined in com.c */
extern int buffersize;
extern int nrfds;
```

```
char recvbuff[MSGSIZE]; /* Buffer to store messages */
char *recvptr=NULL; /* that were received but not */
 /* yet processed... */

#ifdef SOLARIS
struct sockaddr socknm = {
 AF_UNIX,
 “/tmp/.spsck\0”
 };

int len_soadr = sizeof(struct sockaddr);
#else
const struct sockaddr_un socknm = {
 AF_UNIX,
 “/tmp/.spinsocket\0”
 };

int len_soadr = sizeof(struct sockaddr_un);
#endif

struct qstate          /* List of Queue States is kept in each */
{                       /* Client to reduce the Server Load by */
 char name[16];        /* sending notifications to all Clients */
 int filled;              /* if the Data in the Queue changes. */
 struct qstate *nxt; /* So, “busy waiting” with constant server */
};                                 /* Accesses can be avoided. */

struct qstate *qstates; /* Pointer to the Start of this List. */

/* ------------------------------------------------------------ */

struct qstate
*new_qstate(void)
{
 return (struct qstate *) malloc(sizeof(struct qstate));
}

void
modify_qstate(char *name, int filled)
{
 struct qstate *p;

 for(p=qstates; p!=NULL; p=p->nxt)
 {
 if(strcmp(name,p->name)==0)
 {
    p->filled=filled;
    break;
 }
 }
 if(p==NULL)
 {
 struct qstate *pp;

 pp=new_qstate();
 (void) strncpy(pp->name, name, 16);
 pp->filled=filled;

 if(qstates==NULL) /* first one */
 {
    qstates=pp;
 }
 else
 {
    p=qstates;
    while(p->nxt!=NULL)
    {
    p=p->nxt;
    }
    p->nxt=pp;
 }
 }
}

int
check_qstate(char *name)
{
 struct qstate *p;
 int r=0;

 for(p=qstates; p!=NULL; p=p->nxt)
 {
 if(strcmp(name,p->name)==0)
   {
    r=p->filled;
    break;
   }
 }

 return(r);
}

/* EXEMSG handles “@” messages sent by the server, notifying */
/* the client of the actual length of the server queue. */
void
exemsg(char *buffer)
{
 char channame[256];
 int value;
 char msgtype;
 (void) sscanf(buffer,"%c%s %d\n",&msgtype,channame,&value);
 if(strlen(channame)>16)
 {
 channame[16]='\0';
 }

 if(msgtype=='@')
 {
 modify_qstate(channame,value);
 }
 else
 {
 printf("WARNING: Don't know what to do with '%c' Message\n",msgtype);
 }
}

int
sockhasdata(int esd)
{

 int t,i;

 /* The ioctl returns the number of bytes available for reading */
 t=ioctl(esd, FIONREAD, &i);
 if(t==-1)
 {
 printf("Ioctl Error trying to peek the Socket.\n");
 perror("ioctl:");
 exit(1);
 }
 return(i);
}


void
connect_server(void)
{
 int t;

#ifdef SOLARIS
 FD_ZERO(&inputfdmask);
 printf("Trying to connect the Server via AF_UNIX Socket %s.\n",socknm.sa_data);
#else
 printf("Trying to connect the Server via AF_UNIX Socket %s.\n",socknm.sun_path);
#endif

 extsock=socket(AF_UNIX, SOCK_STREAM, 0);
 if(extsock==-1)
 {
 printf("Error creating socket. (%d)\n",errno);
 perror("Error ");
 exit(1);
 }

 t=connect(extsock, &socknm, len_soadr);
 if(t!=0)
 {
 printf("Error connecting to Server. (%d)\n",t);
 perror("");
 unlink("/tmp/.spinsocket");

 exit(1);
 }
 printf("Connected.\n");

 printf("Socket Descriptor is %d\n",extsock);
 FD_SET(extsock, &inputfdmask);
 /* inputfdmask|=MASK(extsock); */
 nrfds++;
}


void
do_recv(int esd, char *buf, int bufsize)
{
 int t;
 char *p;

 if(recvptr==NULL) /* Queue is empty - really receive */
 {
#ifdef SOLARIS
 t=recv(esd, buf, bufsize, 0);
#else
 t=recv(esd, buf, bufsize, NULL);
#endif
 if(t==-1)
    {
     perror("recv ");
     exit(1);
    }
#ifdef DEBUG1
 printf("recv< %s\n",buf);
#endif

 p=memchr(buf,0,bufsize); /* Find end of message */
 recvptr=strchr(p+1,13);
 if(recvptr) /* There is another message after this one */
    {
#ifdef DEBUG1
     printf("Multiple Messages received with one recv() Call.\n");
#endif
     (void) memset(recvbuff, 0, MSGSIZE);
     (void) memcpy(recvbuff, p, MSGSIZE-((int) (p-buf)) );
     recvptr=recvbuff;
    }
 }
 else
 { /* We still have messages that are not yet processed */
#ifdef DEBUG1
 printf(“Processing Message from local Buffer.\n”);
#endif
 p=memchr(recvptr,0,MSGSIZE-((int) (recvptr-recvbuff))); /* End of that mes-
sage... */
 (void) memset(buf, 0, MSGSIZE);
 (void) strcpy(buf, recvbuff);
 if(memchr(p,0,MSGSIZE-((int) (p-recvbuff)) ) )
 { /* There is another message after this one */
     recvptr=memchr(p,0,MSGSIZE-((int) (p-recvbuff)) )+1;
    }
 else
    {
     recvptr=NULL;
    }
#ifdef DEBUG1
 printf(“recv (from local buffer) <%s\n”,buf);
#endif
 }
}

void
do_send(int esd, char *sendbuf, int sendbufsize)
{
 int t;

 t=send(esd, sendbuf, sendbufsize, 0);
 if(t==-1)
 {
 perror(“send “);
 exit(1);
 }
#ifdef DEBUG
 printf(“sent> %s\n”,sendbuf);
#endif
}

/* The following Procedures are called from the modified pan code */
/* and therefore are the “interface” between the ext. comm.    */
/* and the SPIN System. */

/* sendext() just sends out the message to the server. */
void
sendext(char *buffer, char *qname, int qflds)
{
 (void) memset(buffer1, 0, buffersize);
 (void) sprintf(buffer1, “!%s %d %s\n”, qname, qflds, buffer);

 do_send(extsock, buffer1, buffersize);
}


/* recvext requests the server for a message for the given */
/* channel name and then waits for it and receives it */
/* if done==1 it asks the server to erase the data in the */
/* queue, otherwise the queue data structure in the server */
/* is not touched */
int
recvext(char *buffer, char *qname, int qflds, int slot, int fld, int done)
{
 char channame[256];
 int nflds;
 int value;
 char msgtype;

 /* recvext should return just one integer, if done=1 it should */
 /* remove the data from the queue */

 /* First, send a Message to the Server to request the Data */
 (void) memset(buffer1, 0, buffersize);
 if(done==1)
 {
 sprintf(buffer1, “/%s %d %d\n”,qname,qflds,fld); /* with deletion */
 modify_qstate(qname, 0); /* It will be re-set by the next @ msg */
 }
 else
 {
 sprintf(buffer1, “-%s %d %d\n”,qname,qflds,fld); /* peek only */
 }
 do_send(extsock,buffer1,buffersize);

waitforanswer:
 /* Now, wait for the Answer from the Server */
 /* TODO: Implement some Kind of Timeout */

 msgtype=’+’; nflds=0; value=0;
 do_recv(extsock,buffer,buffersize);

 sscanf(buffer,”%c%s %d %d\n”,&msgtype,channame,&nflds,&value);
 if(strlen(channame)>16)
 {
 channame[16]=’\0’;
 }

 if(msgtype!=’!’) /* Something else arrived - may happen ... */
 {
```

```
 exemsg(buffer);
 goto waitforanswer;
 }

 if(strcmp(channame,qname)!=0)
 {
 printf("WARNING: Requested Data for %s but got Reply for %s\n",qname,channame);
 exit(1);
 }

 return(value);
}

int
extqlen(char *qname)
{
 int length;

 /* TODO: Check if we really have read Access to this Channel */
 /* by explicitly asking the Server (or adding the socket */
 /* descriptor in the @mesg that is sent out by the Server. */

 if(sockhasdata(extsock))
 {
 do_recv(extsock,buffer,buffersize);
 exemsg(buffer);
 }

 length=check_qstate(qname);

 return(length);
}
```

B.3: Communication Code for the Server

```
/* $Id: comserv.src.c,v 1.10 1996/04/11 09:34:32 loeffler Exp loeffler $ */

/* EXTERNAL COMMUNICATION ROUTINES, SERVER CODE */

/* This file contains the procedures that are necessary for ext. */
/* communication between the compiled PROMELA code and other */
/* UNIX processes (which may either be compiled PROMELA or */
/* something different). */
/* The code in this file is for a communication server. Only */
/* one server must be present in a distributed environment */
/* The server keeps track of all data in all external Channels */
/* (c) 1995,96 by Siegfried Loeffler */
/* Ecole Nationale Superieure des Telecommunications, Paris */
/* Networks Department / Dept. Reseaux */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* for AF_UNIX */
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>

#ifdef SOLARIS
#include <sys/filio.h> /* for FIONREAD */
#include <stropts.h>
#include <poll.h>
#endif

/* Function Prototypes for external Interface --------------- */
int recvext(char *buffer,char *qname,int qflds,int slot,int fld,int done);
void sendext(char *buffer,char *qname,int qflds);
int extqlen(char *qname);

extern int unlink();
extern void free();
extern char* malloc();
extern char* emalloc();
/* ---------------------------------------------------------- */

int extsock; /* File Descr ref. to socket */

#define MSGSIZE 1024
char recvbuff[MSGSIZE];           /* Buffer to store messages */
char *recvptr=NULL;             /* that were received but not */
                                          /* yet processed... */

char extbuffer[1024];
extern char *buffer,*buffer1; /* defined in com.c */
extern int buffersize;
extern int nrfds;

#ifdef SOLARIS
/* On SOLARIS this MUST NOT be a constant! */
struct sockaddr socknm = {
 AF_UNIX,
 “/tmp/.spsck\0”
 };
```

```
int len_soadr = sizeof(struct sockaddr);
#else
const struct sockaddr_un socknm = {
 AF_UNIX,
 “/tmp/.spinsocket\0”
 };

int len_soadr = sizeof(struct sockaddr_un);
#endif

/* Data Structures for the Linked List of external Channels - */
struct extchan
{
 char name[16];          /* Channel Name (used as Identifier) */
 int nflds;        /* How many Fields for each send/receive ? */
 int filled;     /* How many Data items are currently stored? */
 char databuf[MAXPARS*13];
 int readsock; /* Socket Descriptor that may read the Channel */
 struct extchan *nxt;
};
struct extchan *extchans;

/* Data Structures for the List of external Connections */
struct extconn
{
 int sd;
 struct extconn *nxt;
};
struct extconn *extconns;

/* ------------------------------------------------------------ */

/* initialize_server() creates the AF_UNIX socket on which */
/* the server will listen to connect request from other UNIX */
/* processes */

void
initialize_server(void)
{
 int t;
 struct stat statbuf;

#ifdef SOLARIS
 t=stat(socknm.sa_data, &statbuf);
#else
 t=stat(socknm.sun_path, &statbuf);
#endif
 if(t!=ENOENT) /* File does not exist */
 {
#ifdef SOLARIS
 (void) unlink(socknm.sa_data);
#else
 (void) unlink(socknm.sun_path);
#endif
 }

 extsock=socket(AF_UNIX, SOCK_STREAM, 0);
 if(extsock==-1)
 {
 printf("Error creating Socket.\n");
 perror("socket");
 exit(1);
 }

#ifdef SOLARIS
 printf("Creating new %s File\n",socknm.sa_data);
 t=bind(extsock, &socknm, sizeof(socknm.sa_data));
#else
 printf("Creating new %s File\n",socknm.sun_path);
 t=bind(extsock, &socknm, len_soadr);
#endif
 if(t==-1)
 {
#ifdef SOLARIS
 printf("Error binding to %s.\n",socknm.sa_data);
#else
 printf("Error binding to %s.\n",socknm.sun_path);
#endif
 perror("bind");
 exit(1);
 }
 t=listen(extsock, 5);
 if(t==-1)
 {
#ifdef SOLARIS
 printf("Error trying to listen to %s.\n",socknm.sa_data);
#else
 printf("Error trying to listen to %s.\n",socknm.sun_path);
#endif
 perror("listen");
 exit(1);
 }

 (void) signal(SIGPIPE,SIG_IGN); /* "Broken Pipe" Signal from */
                         /* Clients that have been terminated */

 FD_SET(extsock, &inputfdmask);
 nrfds++;

 printf("Server ready for connections.\n");

 t=fcntl(extsock, F_SETFL, O_NONBLOCK);
 if(t==-1)
 {
 printf(“Error setting Socket to Non-Blocking Mode.\n”);
 perror(“fnctl”);
 exit(1);
 }
}


struct extconn
*new_extconn(void)
{
 return (struct extconn *) malloc(sizeof(struct extconn));
}


void
addextconn(int socktoadd)
{
 struct extconn *p, *pp;

 pp=new_extconn();
 pp->sd=socktoadd;

 if(extconns==NULL) /*First one*/
 {
 extconns=pp;
 }
 else
 {
 p=extconns;
 while(p->nxt!=NULL)
    {
     p=p->nxt;
    }
 p->nxt=pp;
 }
}


void
delextconn(int socktodel)
{
 struct extconn *p,*pp;
 struct extchan *x;

 p=extconns;

 if((extconns->nxt==NULL)&&(extconns->sd==socktodel))
 {
 /* Delete the only external Connection */
 extconns=NULL;
 }
 else if(extconns->sd==socktodel)
 {
 /* Delete the first external Connection */
 pp=extconns;
 extconns=pp->nxt;
 (void) free(pp);
 }
 else
 {
 for(p=extconns; p->nxt!=NULL; p=p->nxt)
    {
     if(p->nxt->sd==socktodel)
     {
     pp=p->nxt;
     p->nxt=pp->nxt;
     (void) free(pp);
     break;
     }
    }
 }

 /* Now, we “unlock” all Channels that were bound to this Client */
 for(x=extchans; x!=NULL; x=x->nxt)
 {
 if(x->readsock==socktodel)
    {
     x->readsock=0;
    }
 }

}


void
do_send(int esd, char *sendbuf, int sendbufsize)
{
 int t;

#ifdef DEBUG
 printf(“send> %s\n”,sendbuf);
#endif

 t=send(esd,sendbuf,sendbufsize,0);
 if(t==-1)
 {
 if(errno==EPIPE)
    {
     /* “Broken Pipe” -> Client has been terminated */
     printf(“WARNING: A Client has been terminated. Discarding Send Queue.\n”);
     delextconn(esd);
     FD_CLR(esd, &inputfdmask);
    }
 else
    {
     printf(“Send Error\n”);
     perror(“send “);
     exit(1);
    }
 }

}


void
check_new_conn(void)
{
 int newsock;
 struct extchan *p;

#ifdef SOLARIS
 struct sockaddr newsockaddr;
#else
 struct sockaddr_un newsockaddr;
#endif

 /* Function for the Server */
 /* See if there are new Connect Requests to /tmp/.spinsocket */

 newsock=accept(extsock, &newsockaddr, &len_soadr);
 if(newsock!=-1)
 {
 /* Connection requested. Add it to the List. */
 printf(“Server: Adding new connection to Connection List...\n”);
 addextconn(newsock);
 /* inputfdmask|=MASK(newsock); */
 FD_SET(newsock, &inputfdmask);
 nrfds++;
 printf(“Connection added.\n”);

 /* The newly connected client must be informed over the actual */
 /* Status of the Channels by sending messages... */
 if(extchans!=NULL)
    {
     for(p=extchans; p!=NULL; p=p->nxt)
     {
     (void) memset(buffer,0,buffersize);
     sprintf(buffer,”@%s %d\n”,p->name,p->filled);
     do_send(newsock,buffer,buffersize);
     }
    }
 }
 else
 {
 switch(errno)
    {
    case EAGAIN:
     break; /* Would block, so there are no requests */

    default:
     printf(“Accept Error #%d\n”,errno);
     perror(“accept “);
     exit(1);
    }
 }
}


void
do_recv(int esd, char *buf, int bufsize)
{
 int t;
 char *p;

 if(recvptr==NULL) /* Queue is empty - really receive */
 {
#ifdef SOLARIS
 t=recv(esd, buf, bufsize, 0);
#else
 t=recv(esd, buf, bufsize, NULL);
#endif
 if(t==-1)
    {
     perror(“recv “);
     exit(1);
    }
#ifdef DEBUG1
 printf(“recv< %s\n”,buf);
#endif

 p=memchr(buf,0,bufsize); /* Find end of message */
 recvptr=strchr(p+1,13);
 if(recvptr) /* There is another message after this one */
    {
#ifdef DEBUG1
     printf(“Multiple Messages received with one recv() Call.\n”);
#endif
     (void) memset(recvbuff, 0, MSGSIZE);
     (void) memcpy(recvbuff, p, MSGSIZE-((int) (p-buf)) );
     recvptr=recvbuff;
    }
 }
 else
 { /* We still have messages that are not yet processed */
#ifdef DEBUG1
 printf("Processing Message from local Buffer.\n");
#endif
 p=memchr(recvptr,0,MSGSIZE-((int) (recvptr-recvbuff))); /* End of that mes-
sage... */
 (void) memset(buf, 0, MSGSIZE);
 (void) strcpy(buf, recvbuff);
 if(memchr(p,0,MSGSIZE-((int) (p-recvbuff)) ) )
 { /* There is another message after this one */
    recvptr=memchr(p,0,MSGSIZE-((int) (p-recvbuff)) )+1;
   }
 else
   {
    recvptr=NULL;
   }
#ifdef DEBUG1
 printf("recv (from local buffer) <%s\n",buf);
#endif
 }
}


int
sockhasdata(int esd)
{
 int t,i;

/* (The ioctl returns the number of bytes available for reading */
 t=ioctl(esd, FIONREAD, &i);

 if(t==-1)
 {
 printf("Ioctl Error trying to peek the Socket.\n");
 perror("ioctl:");
 exit(1);
 }
 return(i);
}

struct extchan
*new_chan(void)
{
 return (struct extchan *) malloc(sizeof(struct extchan));
}


struct extchan
*add_chan(char *channame, int nflds)
{
 struct extchan *pp,*p;

 pp=new_chan();

 strncpy(pp->name,channame,16);
 pp->nflds=nflds;
 pp->filled=0;
 pp->nxt=NULL;

 if(extchans==NULL) /*First one*/
 {
 extchans=pp;
 }
 else
 {
 p=extchans;
 while(p->nxt!=NULL)
    {
     p=p->nxt;
    }
 p->nxt=pp;
 }
 return(pp); /* This returns pointer to the new created */
             /* object in the list, not to the list itself */
}


/* findchan() searches the channel list for the channel with */
/* the given name and returns a pointer to the channels data */
/* structure (if it exists) */
struct extchan
*findchan(char *channame)
{
 struct extchan *p;
 struct extchan *d=NULL;

 if(extchans!=NULL)
 {
 for(p=extchans; p!=NULL; p=p->nxt)
    {
     if(strncmp(p->name,channame,16)==0)
     {
     d=p;
     break;
     }
    }
 }
 return(d);
}
```

```
/* checksocks() is called periodically to check if there is any */
/* data on the sockets to be received. if so, the data is */
/* received and put into the waiting queues to be read by */
/* recvext() */
void
checksocks(void)
{
 int t;
 int i,value;
 int nflds,fldnr;
 char channame[256];
 char msgtype;
 struct extchan *p=NULL;
 struct extconn *s,*ss;

 check_new_conn(); /* See if there are new connections */

 for(s=extconns; s!=NULL; s=s->nxt)
 {
 while(sockhasdata(s->sd))
    {
 msgtype='+'; nflds=0;
     do_recv(s->sd, buffer, buffersize);

     (void) memset(buffer1, 0, buffersize);
     (void) sscanf(buffer,"%1s%s %d %s\n",&msgtype,&channame,&nflds,buffer1);
     if(strlen(channame)>16)
     {
     channame[16]='\0';
     }

     if(msgtype=='!')
     {
         p=findchan(channame);
         if(p==NULL) /* new channel name */
         {
         p=add_chan(channame, nflds);
         }

         if(nflds!=p->nflds)
         {
         printf("Warning: Write-Access to Channel %s with %d Fields, but Defini-
tion as %d Fields.\n", p->name, nflds, p->nflds);
         }

         /* Now write into the Queue: */
         if (strlen(p->databuf)!=0)
         {
         (void) strcat(p->databuf, ",");
         }
         (void) strcat(p->databuf,buffer1);
```

```
        p->filled=p->filled+nflds;
#ifdef DEBUG
        printf(“QCONTENTS %s = %s (filled %d)\n”,p->name,p->databuf,p->filled);
#endif

        /* Send out a notification to all Clients, that */
        /* the contents of this Queue has changed. */

        for(ss=extconns; ss!=NULL; ss=ss->nxt)
        {
        (void) memset(buffer, 0, buffersize);
        sprintf(buffer,”@%s %d\n”,p->name,p->filled);

        do_send(ss->sd,buffer,buffersize);
        }

    }

    else if((msgtype==’-’)||(msgtype==’/’))
    {
        /* Here we have to search for the Channel */
        /* get the Data out of the List, eventually */
        /* delete it (if done=1, that is msgtype “/”) */
        /* and then do a send on the socket on which */
        /* the request was received */

        /* Which Field is asked for? */
        (void) sscanf(buffer1,”%d %s\n”,&fldnr,buffer1);

        p=findchan(channame);
        if(p!=NULL) /* found it ... */
        {
        if((p->readsock!=0)&&(p->readsock!=s->sd))
        {
        printf(“Attempt to Read Channel %s, although it\nwas already read by
another Client Process\n”,p->name);
        exit(1);
        }
        else if(p->readsock==0)
        {
        p->readsock=s->sd;
        }

        if(nflds!=p->nflds)
        {
        printf(“Warning: Read-Access to Channel %s with %d Fields, but Definition
as %d Fields.\n”, p->name, nflds, p->nflds);
        }

        if(fldnr>p->filled)
        {
```

```
        printf(“Fatal Error: Request for Field %d cannot be statisfied, because
Queue \nholds only %d Fields.”,fldnr,p->filled);
        exit(1);
        }

        /* get out of the Queue: */
        (void) memset(buffer1, 0, buffersize);
        (void) strcpy(buffer1, p->databuf);

        for(i=0; i<fldnr+1; i++)
        {
        (void) sscanf(buffer1,”%d,%s\n”,&value,buffer1);
        }

        (void) memset(buffer, 0, buffersize);
        sprintf(buffer,”!%s %d %d\n”,p->name,nflds,value);
        do_send(s->sd,buffer,buffersize);

        if(msgtype==’/’)
        {
        /* remove (nflds) from the Queue */
        if(p->filled>nflds)
        {
        (void) strcpy(buffer1, p->databuf);
        for(i=0; i<nflds; i++)
        {
        (void) sscanf(buffer1, “%d,%s\n”, &t, buffer1);
        }
        (void) strcpy(p->databuf, buffer1);
        p->filled=p->filled-nflds;
        }
        else
        {
        strcpy(p->databuf,””);
        p->filled=0;
        }

        (void) memset(buffer, 0, buffersize);
        sprintf(buffer,”@%s %d\n”,p->name,p->filled);
        do_send(s->sd,buffer,buffersize);
        }

        }
 else
 {
#ifdef DEBUG
        printf(“WARNING: Channel %s not found.\n”,channame);
#endif
        }
        }
```

```
     else if(msgtype=='#')
     {
 /* Search for the Channel and return qlen */
         p=findchan(channame);
         if(p->readsock==0)
         {
         p->readsock=s->sd;
         }
         (void) memset(buffer, 0, buffersize);

         if((p!=NULL) && (p->readsock!=s->sd))
         {
         sprintf(buffer, "*%s %d -1\n", p->name, p->nflds, -1);
         printf("Attempt to Read Channel Length of a Channel that was already read
by another\nClient Process\n");
         }
         else if(p!=NULL)
         {
         sprintf(buffer, "*%s %d %d\n" ,p->name, p->nflds,
 p->filled);
         }
         else /* p=NULL, Channel does not exist... */
         {
         sprintf(buffer,"*%s 0 0\n",channame);
         }

         do_send(s->sd,buffer,buffersize);
         }
    }
 }
}

/* The following Procedures are called from the modified pan code */
/* and therefore are the "interface" between the ext. commu. */
/* and the SPIN System. */

void
sendext(char *buffer, char *qname, int qflds)
{
 struct extchan *p=NULL;
 struct extconn *ss;

 /* This takes the Data from the SPIN Queue and puts it into */
 /* the Servers own queue so that other (client) Processes */
 /* have access to it. */

 p=findchan(qname);

 if(p==NULL) /* new channel name */
 {
 p=add_chan(qname, qflds);
 }

 if(strlen(p->databuf)!=0)
 {
 (void) strcat(p->databuf, “,”);
 }
 (void) strcat(p->databuf,buffer);

 p->filled=p->filled+qflds;

 for(ss=extconns; ss!=NULL; ss=ss->nxt)
 {
 (void) memset(buffer, 0, buffersize);
 sprintf(buffer,”@%s %d\n”,p->name,p->filled);

 do_send(ss->sd,buffer,buffersize);
 }

#ifdef DEBUG
 printf(“QCONTENTS sendext: %s is %s (filled %d)\n”,p->name,p->databuf,p-
>filled);
#endif
}


int
recvext(char *buffer, char *qname, int qflds, int slot, int fld, int done)
{
 int t,i,r;

 struct extchan *p;
 struct extconn *ss;

 /* recvext should return just one integer, if done=1 it should */
 /* remove the data from the queue */

 checksocks();

 p=findchan(qname);
 if(p==NULL)
 {
 printf(“recvext: Channel %s not found.\n”,qname);
 exit(1);
 }

 if((p->readsock!=0)&&(p->readsock!=-1))
 {
 printf(“Emergency Stop:\n”);
 printf(“Attempt to Read Channel %s although it was already read by another Pro-
cess\n”,p->name);
 exit(1);
 }
 else if(p->readsock==0)
 {
 p->readsock=-1; /* Only the Server may read */
 }

 (void) memset(buffer1, 0, buffersize);
 (void) strcpy(buffer1, p->databuf);

 for(i=0; i<fld+1; i++)
 {
 (void) sscanf(buffer1, “%d,%s”, &r, buffer1);
 }

 if(done==1)
 {
 if(p->filled>qflds)
   {
    (void) strcpy(buffer1, p->databuf);
    for(i=0; i<qflds; i++)
    {
    (void) sscanf(buffer1, “%d,%s\n”, &t, buffer1);
    }
    (void) strcpy(p->databuf, buffer1);
    p->filled=p->filled-qflds;
   }
 else
   {
    strcpy(p->databuf,””);
    p->filled=0;
   }

 /* After something has been deleted, all Clients are notified */
 for(ss=extconns; ss!=NULL; ss=ss->nxt)
   {
    (void) memset(buffer, 0, buffersize);
    sprintf(buffer,”@%s %d\n”,p->name,p->filled);

    do_send(ss->sd,buffer,buffersize);
   }
 }

 return(r);
}


int
extqlen(char *qname)
{
 int t=0;
 struct extchan *p=NULL;
 checksocks(); /* See if something new has arrived ? */

 p=findchan(qname);

 if(p!=NULL)
 {
 if((p->readsock!=0)&&(p->readsock!=-1))
    {
     printf(“Attempt to Read Channel that was already read by a Client Pro-
cess\n”);
     exit(1);
    }
 else if(p->readsock==0)
    {
     p->readsock=-1;
    }
 t=p->filled;
 }
 return(t);
}
```

Appendix C: 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

[17] Elie Najm, Frank Olsen, *Reactive SPIN and PROMELA*, in Proceedings of the 1st SPIN Workshop, Montréal, Quebec, 1995

[18] Abrial, Jean-Raymond, *Steam-boiler control specification problem*, http://www.informatik.uni-kiel.de/~procos/dag9523/steam-boiler-problem.ps.Z, 1994

[19] Abrial, Jean-Raymond, *Additional Information Concerning the Physical Behaviour of the Steam Boiler*, http://www.informatik.uni-kiel.de/~procos/dag9523/steam-boiler-additional-info.ps.Z, 1996

[20] Gregory Duval, Thierry Cattel, *Specifying and Verifying the Steam Boiler Problem with SPIN*, Ecole Polytechnique Fédérale Lausanne, Switzerland, 1996