Creating A Validated Implementation Of The Steam Boiler Control

Siegfried Löffler and Ahmed Serhrouchni

ABSTRACT. SPIN is a tool for the simulation and verification of protocols. PROMELA, its source language, is a formal description technique like SDL and Estelle that is based on communicating state machines. The tool and the language are in the public domain and therefore widely used. The "Steam-Boiler Control Specification Problem" consists of an informal specification of a steam boiler system in a nuclear power plant. In this paper we show that PROMELA is suitable for the description of a technical system like the steam boiler. We describe the methods which we used to translate the informal problem description into a PROMELA specification. Further, we present our extensions to the SPIN system, which allow an automatic generation of compiled implementations from PROMELA sourcecodes. We summarise the extensions to PROMELA that we found necessary for the creation of the implementation.

1. The Steam Boiler Control Specification Problem

The "Steam Boiler Control Specification Problem" [1] was given to the participants of the Dagstuhl meeting "Methods for Semantics and Specification" which was organised by Egon Börger (Pisa) and Hans Langmaack (Kiel) in June 1995.

The problem specification was published by Jean-Raymond Abrial and describes a control program which serves to control the water level in a steam boiler by communicating with a set of physical devices. It is based on a real specification by the Ïnstitute for Risk Research" and the Ïnstitut de Protection et de Sureté Nucléaire" and therefore very informal and strongly aimed at a particular implementation.

The specification does not describe implementation details, such as message formats or exact physical behaviour of the components. One of the main goals when trying to translate the informal specification into a formal one should be to find out which details are not described exactly enough.

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 validate the control program with a formal description technique.

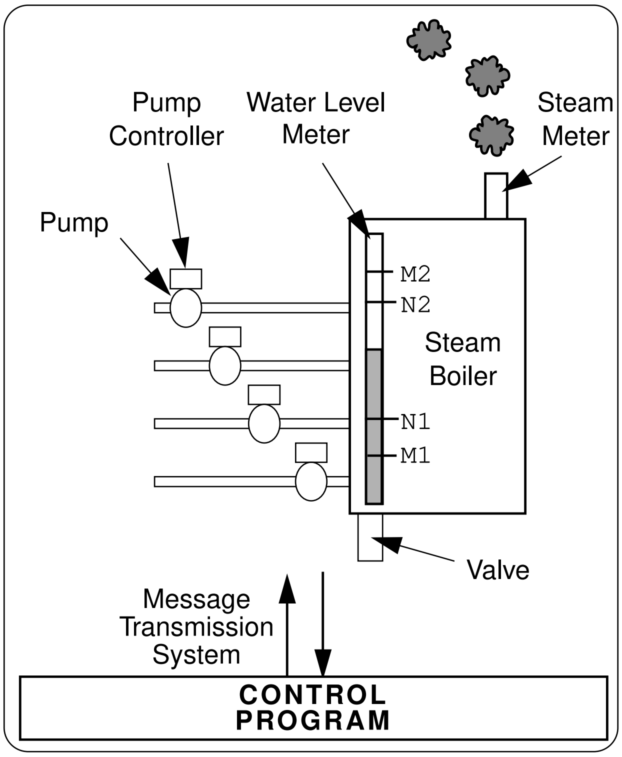

Figure 1. The Physical Structure Of The Steam Boiler

The physical structure of the system is depicted in figure 1. The system comprises the following physical entities:

- The steam boiler itself
- A device to measure the quantity of water in the boiler
- A device to measure the quantity of steam which comes out of the boiler
- A valve to empty the boiler during the initialization phase
- Four pumps to provide the steam boiler with water
- Four pump controllers to supervise the pumps (one for each pump)
- An emergency stop switch (part of the öperator desk", not shown in the figure)
- A message transmission system

The heating (energy supply) for the boiler is not described in the problem specification. We assume that it can not be influenced by the program or any other entity in our system. Therefore the only way to control the water level in the boiler is by switching pumps on or off (except during the initialisation phase, where the boiler can also be emptied using the valve).

For the control program, the task description predefines several modes of operation. The control starts up in the

Initialisation mode: In this mode, it waits for the physical units to become ready and then operates them to achieve a water level between `N1` and `N2`. Then the control program is ready to go into the

Normal mode: in which its task is to maintain the water level in the boiler between the two "normal" levels `N1/N2`. It communicates with the physical units in order to get information about the states of the units (water level etc.) and uses this information to react. The program stays in normal mode until a failure is detected or an emergency stop signal is received.

Degraded mode: In this mode, the program tries to maintain a satisfactory water level despite of the failure of one or more physical units.

Rescue mode: This is the mode in which the program tries to maintain a satisfactory water level in the boiler despite of the failure of the water level measuring unit. This means it has to estimate the water level by a computation which takes the dynamics of the boiler into account.

Emergency stop mode: The control program has to go into the emergency stop mode when either the vital units have a failure or when the water level risks to reach one of its two limit values. This mode can also be reached after the detection of a transmission error between program and physical units. In this mode, the control will just send a command to the physical units in order to stop operation and then terminate itself.

2. Specifying the Steam Boiler System in PROMELA

Other Specifications

The steam boiler control specification problem has already been specified in many different formal description techniques [**2**] [**3**].Compared with those other specifications, the main difference of our model is that it is designed with the goal to create an implementation. The priority is not on a validation which is as exhaustive as possible but on a structure which is as close to reality as possible and therefore allows an easy translation into an implementation.

Development of the Specification

When trying to translate the informal specification into the PROMELA language [**5**] at first we had some difficulties in finding a good starting point. Since the informal specification defines all messages that are sent or received by the control program, we started by defining all those messages using "`#define`" preprocessor directives. Here we found a first problem with the language: PROMELA allows only one "`mtype`" statement for the definition of messages. If we would define all the messages that are used in the steam boiler system in one single statement, the specification would not be very readable — therefore we had to use the "`#define`" statements. This involves the risk of accidentally using the same identifier for two different messages.

After defining the messages, we decided to create one process prototype for every physical entity. This seems to be a good starting point for the description of real-existent systems, as all physical units in reality are independent from each other. In the same way PROMELA proctypes turn in parallel independently. After having created proctypes for all units, we remarked that in some cases this makes the model unnecessarily heavy. For example we do not need to have a separate proctype for the valve since its only task is to empty the boiler during the initialisation phase. This can easily be achieved within the boiler process itself. We also

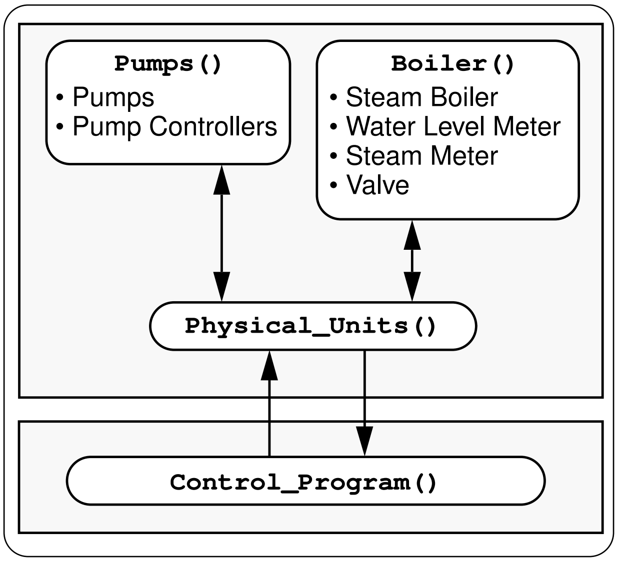

FIGURE 2. The Logical Structure of the PROMELA Specification for the Steam Boiler

integrated the water level measuring device and the steam measuring device into the boiler process and we put all the pumps and pump controllers into one single process.

In order to achieve a better structurisation of the communications, we decided to create a proctype "`Physical_Units`" which does not correspond to a real existent entity in the system. This proctypes task is the coordination of all physical units and the communication between the control program and the units. The control program communicates only with this proctype and never directly with the units. For this communication, only channels and no global variables are being used. As we will see later, this has the advantage that the control program and the set of the physical units can be easily separated in the implementation. Figure 2 shows the final structurisation of the system.

The operating modes which the task description defines for the control program were a very useful guideline for the design of the program. When developing the specification it was possible to start by specifying a control program which contained just an implementation of the initialisation mode. After this mode was successfully implemented and tested, we added the normal mode, which already allowed a simulation of the normal behaviour of the boiler. Then we specified the other modes, however they could not be simulated yet because the specification of the physical units did not yet include possibilities for failures.

So a step–by–step development of the complete control system was possible just by following the well defined structure of the problem description. The resulting specification is quite readable, since it is more or less a "translation" of the informal english text into a formal PROMELA specification.

Real–Time Dependencies

The problem description does not contain much information about the realtime behaviour of the boiler. The only defined time period is the length of the control programs operation cycles, which is five seconds. In those five seconds the control program has the following three tasks:

1. Reception of messages coming from the physical units
2. Analysis of informations which have been received
3. Transmission of messages to the physical units

Because every cycle starts with a reception of messages and ends with a transmission of messages, those messages can be used to synchronise the control program and the physical units.

We decided to do all calculations for the dynamics of the boiler in periods of the length of this cycle. In our model the water level in the boiler does not change continuously but it "jumps" every five "virtual seconds". This is an abstraction from a real existing system. However it is valid to simplify the system like this since for the validation only the maximum behaviour of the boiler is of interest, and this is specified by our model.

Validation

With our current PROMELA model we were only able to verify that the water level in the boiler never goes over `M1` or under `M2`, i.e. that the system can never be damaged. Unfortunately the state space is too big for an exhaustive validation on our machines, so we could just verify the safety properties with the supertrace algorithm and with simulations.

The problem when using simulations to validate the system is that the system is allowed to terminate in case of an emergency stop. Therefore it is not possible to start a simulation and then wait for some days to see if the system works correctly, since most simulations will terminate quite fast.

An improved validation coverage could be achieved by putting more statements in atomic sequences, for example by converting the specification into a "Reactive PROMELA" [**8**] model.

3. Creating the Implementation — Our Extended SPIN Tool

Concept for the Compilation

The generation of executable code from a formal specification is nothing new. Many efforts have been dedicated to this task in the past. Most of these efforts take the formal specification as a starting point and use an abstract tree like the one shown in figure 3.

Usually different tools are used for simulation, validation and creation of the implementation. In Holzmann's PROMELA/SPIN the same tool is used for the simulation and the creation of the validator. Nevertheless, even with PROMELA/SPIN — although the same tool is used — the code that deals with the simulation is quite different from the code for the generation of the validator.

In order to keep as close as possible to the validated code, we did not only use the same tool for the creation of the validator and the implementation but

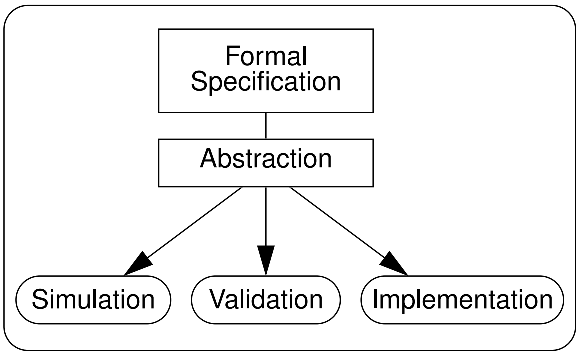

FIGURE 3. Standard Abstract Tree For FDT Tools

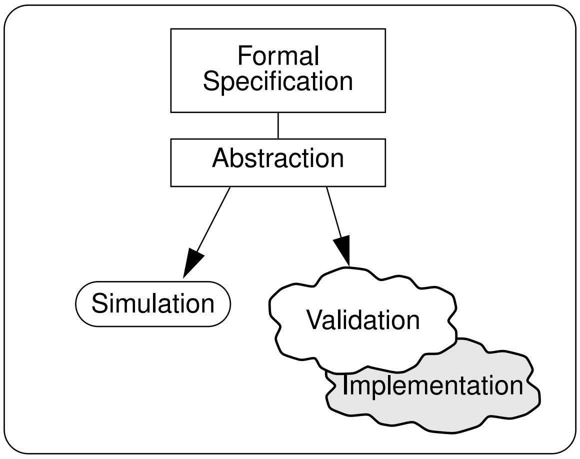

FIGURE 4. Using The Same Tool For Validation And Implementation

we went into the internal data structures of the validator in order to reuse parts of it [9]. We took SPIN and isolated the state machine and transition table — the "motor" of the validator — and extended it with additional code. Doing so, the implementation has a high fidelity to the validated code. The two branches of the abstract tree melt together into one, as shown in figure 4. An overview over the data dependencies in the resulting extended PROMELA/SPIN environment is given in figure 5.

When SPIN is called with the option"`-a`" in order to generate the analyser, it generates a state machine coded in C which corresponds to the PROMELA source.

This is our starting point for the generation of the implementation. In order to re–use this state machine, some kind of scheduler that chooses the executable transitions and executes the corresponding moves is needed in addition to the code from the files "`pan.m`" (the forward moves) and "`pan.t`" (the transition matrix), which is used almost unmodified..

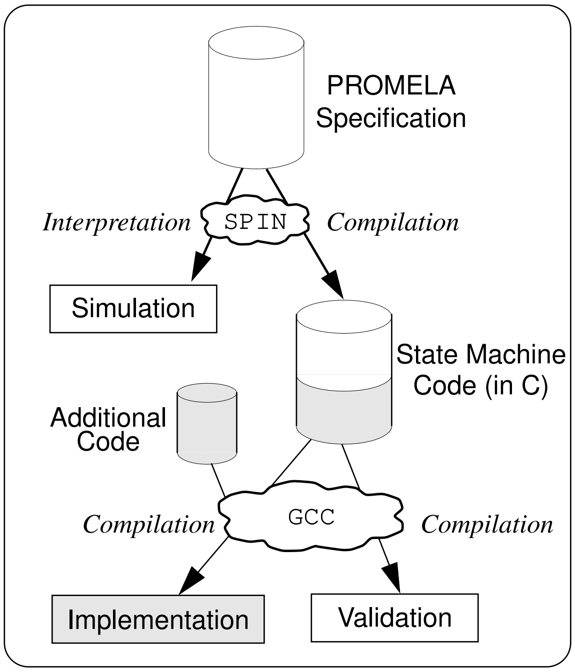

FIGURE 5. Code Dependencies With SPIN

Scheduling in the Implementation

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

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

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

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

of all branches that it already tried to execute because an ëlse" statement that might be one of the branches is only executable if there are no other possible transitions. Since ëlse" 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.

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

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

Atomic Sequences. In PROMELA, sequences of statements may be defined as "`atomic`" sequences. An atomic sequence should, from the point of view of the other proctypes, be seen as one single instruction that is not interruptible. During the execution of an atomic sequence, the scheduler must not switch to another proctype. However, starting with Version 2.0 of SPIN it is legitimate for an atomic sequence to block[**6**] [**7**]. In this case it should — since Version 2.0 — be allowed to switch to another proctype.

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

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

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

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. Care has to be taken as long as the semantics are not formally specified.

External Communications

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

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

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

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

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

Upon compilation of a PROMELA model, one can specify whether the compiled UNIX process should be the server or a client by setting the appropriate switch. Since all channel queues are kept within the server, it is advisable to make the UNIX process the server which uses the external channels the most. For the steam-boiler however this does not really matter because both, control program and physical units, access those channels about the same number of times.

4. The Generation of the Implementation

Our modified SPIN tool now gives us the possibility to execute the state machine which corresponds to our PROMELA model. For the steam boiler control, we can imagine two applications:

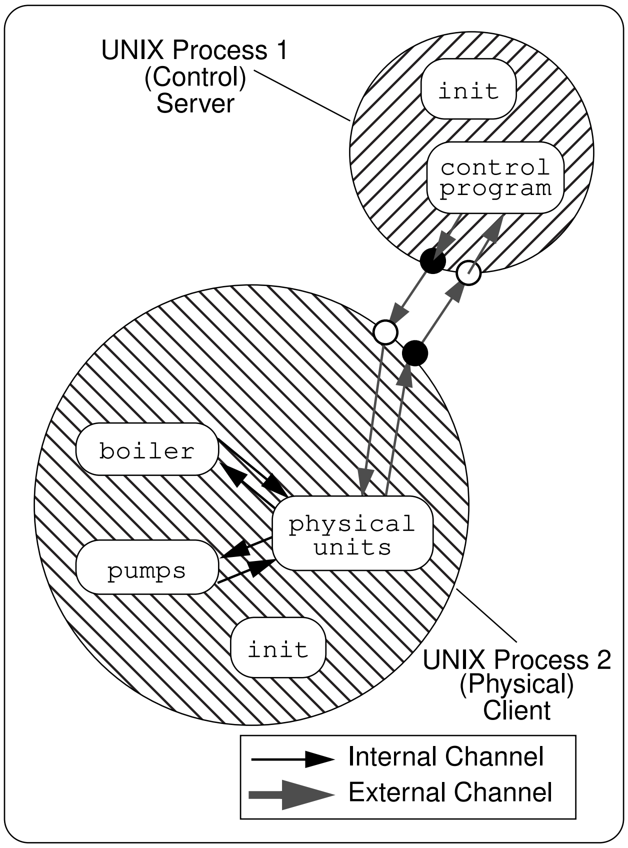

FIGURE 6. The Structure After Dividing The System Into Two Proctypes

1. A stand alone implementation that includes a simulation of the physical entities, or
2. an implementation that divides the model into two UNIX processes, one for the simulation of the physical entities, and one for the implementation of our control program.

Stand–Alone Implementation

In a first step, we translate the same PROMELA source code that we used into a single UNIX process. Except for the scheduling of the active proctype and next transition, this compiled version of the specification behaves exactly like the simulation. This implementation can again be used to validate the model by simulating it. Since it can not communicate with its environment, it can however not be used to control an external boiler simulation or a real boiler. This implementation runs as only one UNIX process which represents the control program as well as the physical units, which is not yet really what we had in mind as an implementation.

Distributed Implementation

For the creation of a more realistic implementation, we need to divide the control program proctype from the parts of the specification which describe the behaviour of the physical units of the system. This means, we have to create two separate UNIX processes, one for the control program and one for the simulation of the physical units (which could then later be replaced by a process that really controls physical units).

This can be achieved with the external channel mechanism which we created for the extended SPIN environment. Now we see why it was a good idea to use only communication channels and no shared memory for the communication between the control program and the physical units. This clear structurisation now makes the separation into the two parts really easy. We just have to define the channels that are used to interconnect the two proctypes as "external" and add two "`init`" proctypes for the UNIX processes that are to be created. Of course the global definitions of messages etc. have to be the same for the two specifications. Figure 6 shows the design of the final distributed implementation.

A problem one might be able to detect when creating such a distributed implementation is the (accidental) use of global variables within both halves of the specification. If such variables are used in a model, they are now local in both halves, and therefore they have to be declared in both of the two specifications. If they are not declared in both of them, the attempt to compile the model will result in an error message, which allows the detection of such accidental use of shared variables.

5. Future Work

The extended SPIN environment

Both, our extended SPIN environment and our specification of the boiler, are not finished yet by far. The most important drawback in our extended SPIN environment is that it is not yet possible to create distributed implementations which use synchronous channels for their interconnection. Although something like a "synchronous channel" cannot really exist in a physical system (since messages cannot travel faster than the speed of light), synchronous channels are a useful language feature that can make specifications much simpler.

Almost as important is the missing capability for sorted send and random receive operations, which were added to SPIN with version 2.0 [**6**] [**7**]. Also introduced with SPIN Version 2.0 and not yet supported are some other constructs like the "`d_step`" statement.

It also has to be mentioned that the compiler itself is not validated, and most probably still contains quite a lot of bugs. Therefore the resulting implementation is not 100% provable error-free.

Another thing that is left to do is a validation of the protocol that we use for the connection of clients and servers.

The steam-boiler model

The steam-boiler model in its current state does not yet cover the whole problem. Additional verification criteria have to be added and additional failure causes have to be implemented in the physical units in order to get a more realistic model.

Much of the code of the current specification could be placed inside "`atomic`" constructs, which would eventually allow an exhaustive validation of the model.

For the generated implementation, it would be nice to have a connection with the Tcl/Tk user-interface by A. Lötzbeyer [**4**]. A first step in this direction has been done by providing C routines that can be used to communicate with compiled implementations.

6. Conclusions

We found that the PROMELA language is not only suitable for the description of communication protocols, but also for the description of a technical system such as the steam boiler and its control. In order to describe such a system, we propose the technique of describing each entity in the system by one proctype and thereafter to integrate multiple proctypes. It is very helpful to create a clear structurisation of the problem before starting to model the system in PROMELA.

Difficulties with PROMELA result from the possible interpretations of its semantics. For the design of implementations, the language should perhaps be refined to describe external events. Further, a possibility to describe realtime timers would be desirable.

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

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

References

[1] J.-R. Abrial, *Steam Boiler Control Specification Problem*, 1994
`http://www.informatik.uni-kiel.de/~procos/dag9523/steam-boiler-problem.ps.Z`
[2] *Solutions for the Steam Boiler Control Specification Problem*,
`http://www.informatik.uni-kiel.de/~procos/dag9523/steam-boiler-solutions.html`
[3] Gregory Duval, Thierry Cattel, *Specifying and Verifying the Steam Boiler Problem with SPIN*, 1996
[4] Annette Lötzbeyer, *A Tcl/Tk-based Steam Boiler Simulator*
`ftp://ftp.fzi.de/pub/korso/steam_boiler`
[5] Gerard J. Holzmann, AT&T, *Design and Validation of Computer Protocols*, Prentice Hall, 1991
[6] Gerard J. Holzmann, AT&T, *What's New in SPIN Version 2.0*, In "SPIN Documentation", 1995
`http://netlib.att.com/netlib/att/cs/home/holzmann-spin.html`
[7] Gerard J. Holzmann, AT&T, *V2. Updates*, In "SPIN Documentation", 1995
`http://netlib.att.com/netlib/att/cs/home/holzmann-spin.html`
[8] E. Najm, F. Olsen, Télécom Paris, *Protocol Verification with Reactive PROMELA*, 2nd SPIN Workshop, 1996
[9] S. Löffler, A. Serhrouchni, *Creating Implementations from PROMELA Models*, 2nd SPIN Workshop, 1996
[11] S. Löffler, A. Serhrouchni, *Protocol Design - From Specification To Implementation*, Proceedings of the 5th Open Workshop on High Speed Networks, Télécom Paris, 1996
[10] S. Löffler, *A Promela To C Compiler, Rapport de Stage*, Télécom Paris, 1996

Siegfried Löffler, Ecole Nationale Superieure des Télécommunications, 46 Rue Barrault, 75013 Paris, France

Current address: Siegfried Löffler, Möhringer Str. 96, 70199 Stuttgart, Germany

E-mail address: fl@lf.net

Ahmed Serhrouchni, Ecole Nationale Superieure des Télécommunications, 46, Rue Barrault, 75013 Paris, France

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