# The Application of Promela and Spin in the BOS Project (abstract)

Pim Kars
Formal Methods Group
Dept. of Tele-Informatics and Open Systems
Faculty of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
kars@cs.utwente.nl

July 11, 1996

**Abstract**

After a short introduction to the BOS project, we discuss the "why, what and how" of the use of formal methods in the project, some recent experience using Promela/Spin and reflections on the pragmatics of validation.

## 1 Introduction

Last year we reported at this workshop on the use of Promela and Spin in the BOS project [Kar95]; this time we will tell a bit more about the background of the project, the use of formal methods and our latest experience: the validation of one of the interfaces of the BOS system. Most papers on validation tend to discuss either theory (algorithms, logics) or case studies showing how a system was modelled and validated. This leaves some interesting issues underdeveloped: the pragmatics and methodology of the validation tool/technology. We want to share some reflections on these issues in the last section.

## 2 The BOS project

The BOS project is part of the keystone of an effort to protect the low, western part of the Netherlands from storm floods. A movable storm surge barrier is currently being built near The Hook of Holland in the Nieuwe Waterweg, the canal connecting the harbour of Rotterdam to the North Sea. The barrier will operate fully automatically because human intervention was deemed too error prone. BOS, an acronym for "Beslis- en Ondersteunend Systeem" (Decision and Support System), is the central system that decides when and how the barrier has to move, and that controls its movements. In order to perform these tasks, the BOS system has to gather measurements of water levels, tidal info, weather data and predictions etc. To this end it is connected to several sensors, either directly or via networks. Furthermore, it includes a software module (provided by a third party) that predicts water levels based on the various measurements. The BOS system has to comply to very high reliability requirements expressed as failure rates of undesirable events, such as not closing the barrier when called for.

The Ministery of Public Works awarded the contract to build the BOS system to Computer

Management Group (CMG). It was clear from the outset to project management at CMG that extra measures were needed in this project given the high reliability requirements. CMG decided that formal methods might be the answer, as is also highly recommended by the IEC 65A norm for highly reliable systems. So CMG set up a close collaboration with the Formal Methods Group at the University of Twente. The idea was to try to integrate formal methods in the design trajectory using the experience and knowledge of both the project team at CMG and our group. (As an aside, the use of "we" in this paper therefore generally includes both the BOS project team at CMG and the Formal Methods group at the University of Twente.) This may be contrasted to an approach where for example validation tasks are performed off-line.

## 3 Formal methods in the BOS project

### 3.1 Choice of methods

It has been said by many authors before: think about why, what and how you want to formalize. One should not be overambitious; it is not feasible in practice to formally specify a large system and, for example, use refinement techniques to get to an implementation. Another reason to be cautious about the goals you have with introducing/integrating formal methods into an existing design trajectory, is that it is risky to introduce too many new techniques at one time into a project.
Early on it was decided that our effort would concentrate on two tasks:

- formally specifying the design, primarily to avoid ambiguities and incompleteness,
- validating crucial parts of the design, in particular the interfaces with the outside world, to get more confidence in the correctness of the mechanisms chosen in the design.

As to the choice of methods, for validation Promela/Spin was a natural choice: we already had some experience using Spin, it is able to handle large state spaces, and the language and tool are relatively easy to learn and use. The last factor was particularly important since the desire to integrate formal methods meant that engineers at CMG would do most of the actual work. The choice of a specification language for the design was harder. One of the limiting conditions was the ability to specify control as well as data aspects. No language that we had experience with completely fitted the bill. The solution was to use two languages: Promela (which already had to be learnt for the validation) for the control part and Z, embedded in Promela, for the data part.

As reported at last year's Spin Workshop, experience with Promela/Spin was positive. The use of Z turned out to be less simple than expected. More on this subject will be reported elsewhere; see also [Wij96].

### 3.2 Integration

Formal methods are only one means to enhance the quality of the system. As such they should be integrated with other measures. Starting from a functional specification, the strategy to get to an implementation was set up roughly as follows:

1. Basic design: decomposition of the functionality into subsystems,

2. Function Failure Analysis to determine critical functions,
3. Detailed design per subsystem:
    - specification in Z and Promela
    - validation of critical parts with Spin
    - selective prototyping
    - review of the design
4. Implementation in C++ and extensive testing by separate teams.

At the moment, the design is almost finished and implementation is under way. The detailed design is described in a set of documents, one per subsystem, containing

- Dutch text for descriptions and explanations,
- Promela for specifying I/O and control (cf. CSpec) structure,
- Z for specifying data (DD) and operations on data (cf. PSpec),
- Data Flow Diagrams for documenting the overall structure.

A case tool (SDW) is used to link the elements in order to maintain consistency and the relation to the Functional Specification.

## 4 Recent experience

### 4.1 BBI, the BOS-BESW interface

Our latest endeavour was the validation of one of the most crucial interfaces: the interface between BOS, the system that takes the decision to move the barrier, and BESW, the system that performs this task. BESW is an acronym for “BEsturingsSysteem Waterweg” (Control System Waterweg). It consists of a separate network of controllers (PLCs) that control the actual movements of all parts of the barrier.

The interface between BOS and BESW is called BBI. Originally this interface was designed as a simple Question-Answer service built on top of a reliable, duplicated data link service, the details of which were left to be filled in later. BOS, acting as the master, sends a command to BESW and BESW answers with an acknowledgement, or BOS sends a status request and BESW answers with the requested status. Apart from reliably transmitting commands, status requests and replies, the interface also provides a “heartbeat” mechanism such that failure of the BOS system or both links to it can be detected by BESW. If such an unlikely event occurs, BESW switches to manual operation. When the time came to fill in the details a setup was chosen in which BOS remotely accesses registers within BESW: bits in these registers are set to signal events.

### 4.2 Validation of the BBI

We will not go into much technical detail of the interface and the model, but concentrate on observations of the validation process.

It was clear from the start that the interface would not be free of errors. In fact, some recent changes in the requirements had made the design incomplete at some points. So our task was not to try to show that it was correct, but rather to find enough severe errors to invalidate the design.

To be able to model the system for validation, specifications of both sides of the interface were needed. The BOS side of the interface was well-documented in a mixture of Promela and Z augmented with informal explanations. The parts relevant to the validation could be translated into Promela with rather small effort. The documentation of the BESW side of the interface was harder to interpret; some but not all of the scenarios were explained by a kind of timing diagrams. Analysis of the BESW side uncovered several unspecified situations, mostly to do with combinations of events, that had to be resolved. Subsequent discussions with the company building BESW showed that indeed these situations would lead to unwanted behaviour including deadlocks! Also enough details of BESW's behaviour became known to finish the model and start simulation and validation. It is interesting to note that most errors were found at the analysis stage prior to validation. Some errors were found by simulation and only a few errors during validation. At this point however our task was already successful in the sense that sufficient errors were found to try to convince management that the interface should be redesigned. Moreover, validation became more and more unmanageable as the model had to be changed each time to fix or mask errors that had been detected.

## 4.3 Some modelling issues in the BBI model

The processes relevant to the BBI were straightforwardly modelled as proctypes connected through asynchronous channels. The processes typically operate in cycles where each cycle consists of a choice between `atomic` steps. The BOS side was kept as close to the specification as possible to minimize translation errors. However some "expensive" mechanisms were simplified. For instance, the BOS process directly connected to the BOS side of BBI uses polling requests to wait for completion of an issued command. In the model this was replaced by waiting for a variable to reach a certain value.

Since Promela has no direct support for modelling time (apart from explicitly modelling clocks) the BBI model was made asynchronous. An advantage of this approach is that the model is less sensitive to deviations in timing. A major drawback however is that the model allows behaviour that "in reality" cannot occur. So each time an error is reported, one has to go back to the original design and check whether the error is really an error in the design or merely a consequence of abstracting from time. Many of the errors reported turned out to belong to the second category, especially when simulation was used. Several solutions can be envisaged to this problem:

- use a timed extension of Promela like [TC96],
- constrain the behaviour by introducing extra synchronizations between processes, e.g. by handing out tokens to regulate the relative speed of processes, using provided clauses to proctypes or using timeout to implement clock cycles,
- constrain the behaviour in the never claim (only applies to validation, not to simulation).

In the BBI model we needed to model a register with several bits, say 8 of them. A simple way to express this in Promela is to use an array of bits:

```
bit command_reg[8] ;
command_reg[5] = 1
```

However, a more compact representation is obtained using plain bytes to model registers and bit-operations on bytes:

```
byte command_reg ;
command_reg = command_reg | (1 << 5)
```

Experiments showed that the second representation not only uses less memory, it also runs faster. The choice of representation may of course be hidden by macros (`#define`s and `#ifdef`s).

In many cases looping through the bits of a register was best considered as a new primitive action modelled as a `d_step`. Unfortunately this was not always possible since Spin currently does not allow a `d_step` within an `atomic` sequence.

## 5 Reflections on Spin pragmatics and methodology

Some case studies give the impression that validation is just a matter of translating a system into a model and push a button. If the result is OK then go ahead and implement the system, otherwise fix the design error and reiterate. An underlying assumption seems to be that it is clear what the system and validation criteria are; that they are given and fixed once and for all. Obviously this is something to strive for, but it is our experience that in reality things are not as simple as that. Gerard Holzmann [Hol91] gives some advice on how to model for Spin validation. We would like to add some comments based on our experience, and outline some of the problems a validator faces when using Promela/Spin (or probably any validation system for that matter).

### 5.1 The validation process

In order to validate a design, be it in Promela/Spin or another tool, several steps must be taken:

1. formalize the design, i.e. translate it into the modelling language,
2. verify the formalization with respect to your intention,
3. formalize the validation criteria, i.e. translate them into the validation language (Promela never claims, LTL formulae),
4. validate the design against the criteria.

It is often assumed that step 2 is best done by simulation and the last by validation. However, there are cases where the two cannot really be separated. In fact, each time a validation run reports an error, the cause may either be in the model (modelling error) or in the original design. Just as the software development process is better described by the spiral model than the waterfall model, here also the steps are not to be considered as discrete steps followed sequentially, but more as a cyclic activity.

## 5.2 Formalization of the design

It may be useful to first indicate the kind of systems we have experience with. The systems we have been validating are mostly interfaces of the BOS system with the outside world. Typically three (or more) entities can be discerned:

- the subsystem being modelled,
- the environment (part of the outside world, i.e. outside of the BOS system), and
- the inner environment: the rest of the BOS system as far as relevant for the validation.

These entities can usually be faithfully modelled as processes communicating asynchronously. The subsystem being modelled is already formally specified in Promela and Z and ideally the model is kept close to this specification. The environment has to be formalized based on documents provided by the supplier. The inner environment has to be abstracted from the relevant specifications. Of these three, the subsystem is really the model in a strict sense. The other two entities are only there to "drive" the model of the subsystem; it is often useful to combine them into a common "driver" process to guide and check end-to-end behaviour, cf. the notion of an upper tester.

There are several circumstances that complicate the formalization such as

- the environment may have several failure modes,
- the behaviour of the environment may not be fully specified, requiring further analysis to complete its specification, and
- the inner environment may be scenario-based.

### Modelling issues: structure and use of language elements

Usually there is more than one way to express something in a language, be it a specification language or a system implementation language, and Promela is no exception to this rule. (If some aspects of the system cannot be represented in the modelling language, an abstraction has to be made rightaway.) As an example see the discussion on modelling registers in section 4.3. As in system implementation languages, issues here seem to be clarity and flexibility versus compactness of code and speed of execution.

Some simple practical advices can be given. It is a good idea to start thinking about how to include the failure modes in the beginning. This means that the model should be structured in such a way that changes in the failure behaviour are easily made. E.g. by macros (`#define`s) with different bodies depending on a global failure-level also indicated by a `#define`.

Also, it is important that the model is structured in such a way that events important to the validation are readily observable. For example these could be read or write events in channels, the change of a global variable (should be global, otherwise it is not observable to a never claim) or reaching a certain state label. These events and their representation should be decided upon early on, because the expression of validation criteria depends on them.

### Level of abstraction

Preferably, the model is kept as close to the original system specification as possible to minimize translation errors. As is well known, however, a complex system may be expensive to validate. Therefore a key issue in modelling a system for validation is to reduce the complexity by abstracting away from irrelevant details (see e.g. [Hol91, section 11.8]). But what are irrelevant details? This depends on the part of the system one is interested in and the (kind of) errors one is looking for. By the way, any validation effort in the Spin (and others) fashion could be said to only be able to reveal expected errors, or at least of an expected kind.

Also the link with the original system needs to be maintained. At one extreme, abstracting away too much detail may result in a model with no errors and hardly any relationship with the original system to be validated. More remarks on this issue follow below.

Summarizing, we have the following constraints on the model

1. should have a close, known relation to the intended system; preferably this relation should also be explicit, well-documented,
2. should be concrete enough to model all system peculiarities,
3. should be small enough to validate in an exceptable time.

### Traceability

A serious problem is how to maintain the link between the (possibly informal) system you wanted to validate and the model that is eventually validated. When errors are found in the model, the original design may be adapted, or the model is adapted to accomodate to the whims of the modelling language, etc. A first observation is that the development of a model may thus be compared to the development of code. In a serious software development project, where the *traceability* of developments and decisions is of utmost importance, it should therefore be put under version control.

Secondly, this seems to demand an approach where the system is specified in a rather concrete fashion, and next this concrete specifictaion is used both for validation and implementation:

- reduce/project the specification for validation,
- refine the specification for implementation.

A reduction strategy, which in our experience can be useful, is to project on (i.e. "hide") syntactical entities like variables, processes, mtypes and channels. A simple strategy could just replace these entities by "don't care" entities; a more refined one would have to use some kind of abstract interpretation, an area of much current research. Tool support for these kind of reductions under user guidance would be most welcome. This as an alternative to manual reduction and prove that the result is equivalent to the original, for example using a tool like `spine` [Erd95]. Or to reduction strategies implemented *inside* the tool such as partial-order reduction as already implemented in Spin (based on the class of properties expressible in nexttime-free linear temporal logic) or dedicated reduction strategies based on the specific property that is being checked.

## 5.3 Validation criteria

Sometimes the impression is given that the criteria are completely known and fixed. This is the case for example if a protocol is modelled for which the intended service is clear. If this service can be faithfully translated into the validation language all is simple and well. More often than not the criteria are made up during validation, can not be fully expressed in the validation language, or the validation language is too abstract, too remote from the validator's frame of mind, that errors are made during the translation of the informal requirements. Regarding this last issue, a simple extension of the LTL interface of Xspin, allowing the expression of combinations of temporal operators such as `precedes`, would be welcome.

Note that Promela/Spin does not clearly distinguish between the model and the validation criteria. For example, `assert`s, monitor processes and various labels are added to the model instead of being held separate.

## 5.4 Validation

A validator using (X)Spin is confronted with many choices. First of all, there are the Spin system parameters such as the maximum memory size, the estimated state space size, the maximum search depth, whether partial-order reduction is used or not, and whether exhaustive or super-trace search mode is used. These parameters mainly determine the performance/coverage of the system. Guidance on how to choose them is provided on one of the help screens of Xspin. Next, there are Spin parameters that determine how many errors are reported and the kind of property checked: safety properties, non-progress cycles, acceptance cycles, or a property expressed as a never claim. These, together with the validator's choices of

1. channel sizes and other constants,
2. which failure mode of the environment is included (semantics of timeout, message loss, node failure, etc.),
3. which specific property or combination of properties is put into the never claim,
4. which scenario or scenarios is exercised, and
5. when the system has multiple layers: the order in which the layers are traversed,

determine a many-dimensional landscape. The validator's itinerary through this landscape has to be chosen judiciously in order to get the most out of validation in the shortest time.

Regarding failure modes, it seems wise to start with a nice environment and add failures gradually. If the design has to be changed, one may have to check again, to be sure that it also works correctly without the added failure behaviour. Theoretically it is possible that for example progress is only assured thanks to this failure behaviour.

Regarding multiple layers, reduction techniques "outside" the tool (see section 5.2) may be useful when a layered system is validated. One can work from the bottom up. After validating a layer, it may be reduced to a simpler model, cf. the `pftp` example in [Hol91]. An alternative is to replace the lower layer by its most general behaviour and put the validated properties as assumptions into the never claim.

### 5.5 Final remarks

Validation is not an end in itself but embedded in a larger activity: the production of a correct system implementation. The objective of validation is usually to get confirmation that a system is indeed correct, or ar least to get more confidence in the correctness, since full correctness will in general not be reachable in practice. The opposite also occurs: in the BBI case the objective was to find sufficient errors to invalidate the design.

A seemingly trivial but interesting question is when to stop validating; the same question also arises in the context of testing. The answer depends on the objective and whether the validation criteria are complete and fixed beforehand. If the objective is to show correctness using fixed criteria and no errors are found, validation can be considered done within the limits of the language/tool. If the objective is to get as many errors as possible (as in the BBI case), or the criteria are not fixed, it depends on when one is satisfied with the results and the trade-off between the effort spent on further validation and the gravity of the errors searched for.

## References

[Erd95] Hakan Erdogmus. Verifying semantic relations in SPIN. In Grégoire [Gré95].

[Gré95] Jean-Charles Grégoire, editor. *Proceedings of SPIN95, the First Spin Workshop*, URL `ftp://netlib.att.com/netlib/spin/ws95/spin95_abstracts.html`. 1995.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[Kar95] Pim Kars. Experience using Spin and Promela in the design of a storm surge barrier control system. In Grégoire [Gré95].

[TC96] Stavros Tripakis and Costas Courcoubetis. Extending PROMELA and Spin for real time. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of TACAS'96*, pages 329–348, 1996. LNCS 1055, Springer.

[Wij96] Klaas Wijbrans. The BOS project: practical experiences with formal specification. In Rom Langerak, editor, *Proceedings of the Third Dutch Specification Day*, 1996.