Software Engineering with Formal Methods : The Development of a Storm Surge Barrier Control System

Seven Myths of Formal Methods Revisited

Jan Tretmans
University of Twente
Formal Methods and Tools group
P.O. Box 217
NL-7500 AE Enschede
tretmans@cs.utwente.nl

Klaas Wijbrans, Michel Chaudron
CMG The Hague B.V.
Division Advanced Technology
P.O. Box 187
NL-2501 CD The Hague
klaas.wijbrans@cmg.nl
michel.chaudron@cmg.nl

1 Introduction

Bos is the software system which controls and operates the storm surge barrier in the Nieuwe Waterweg near Rotterdam. It is a complex, safety-critical system of average size, which was developed by CMG Den Haag B.V., commissioned by Rijkswaterstaat (RWS) – the Dutch Ministry of Transport, Public Works and Water Management. It was completed in October 1998 on time and within budget.

CMG used formal methods in the development of the Bos software. This paper discusses the experiences obtained from their use. Some people claim that the use of formal methods helps in developing correct and reliable software, others claim that formal methods are useless and unworkable. Some of these claims have almost become myths. A number of these myths are described and discussed in a famous article: *Seven Myths of Formal Methods* [Hal90]. The experiences obtained from using formal methods for the development of Bos will be discussed on the basis of this article. We will discuss to what extent these myths are true for the Bos project.

The data for this survey were collected by means of interviews with software engineers working on the Bos project. These include the project manager, designers, implementers and testers, people who participated from the beginning in 1995 until the end in 1998 as well as engineers who only participated in the implementation phase, and engineers with and without previous, large-scale software engineering experience.

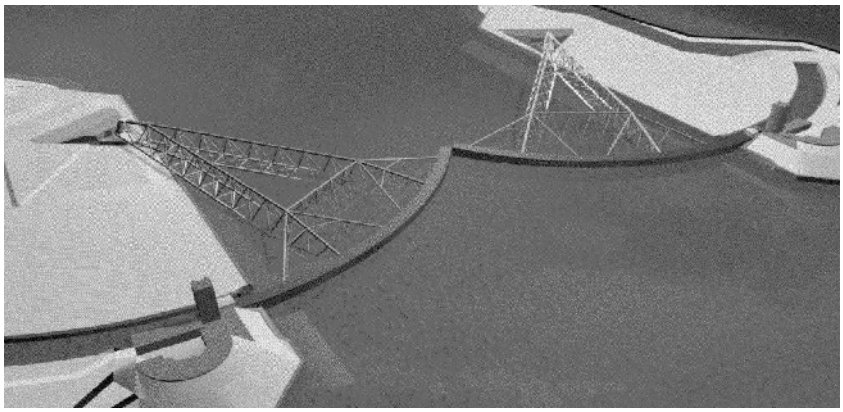

Figure 1: Computer drawing of the *Maeslant Kering*, seen from the sea-side. The width of the waterway is about 300m.

This paper concentrates on the experiences of the software engineers with formal methods. These experiences, placed in the context of the seven myths, are described in section 3. This paper does not discuss technical details about the particular formal methods used or the way they were used; see [Kar97, Kar98] for these aspects. Moreover, formal methods were only one technique used in the development of Bos. The overall engineering approach and the way different methods and techniques were combined to assure the required safety-critical quality, are described [WBG98, WB98]. Testing in Bos is described in more detail in [GWT98], while [CTW99] will give a more systematic analysis of the results of the interviews with the developers.

2 The Bos System

The Storm Surge Barrier The Netherlands are located in a low delta near the sea, into which important rivers such as the Rhine and IJssel flow. The history of our country has been shaped by the struggle against the sea. The great flood disaster of 1953 in Zeeland was a rude shock to the Netherlands, demonstrating yet again that the country was vulnerable. It was shortly after this flood disaster that the Delta Plan was drafted, with measures to prevent such calamities from occurring in the future. This Delta Plan is a defence plan involving building a network of dams in Zeeland and upgrading the existing dikes to a failure rate of 10^{-4}, i.e., one flooding every 10,000 years. The realization of the Delta Plan started soon after 1953 and in 1986 the impressive dam network in Zeeland was finished. The weak point in the defence was now the Nieuwe Waterweg. The Nieuwe Waterweg connects the main port of Rotterdam with the North Sea, hence it is an important shipping route. At the same time, being completely open, it is a major risk for flooding of Rotterdam, since large parts of Rotterdam are situated below sea level. Moreover, the Nieuwe Waterweg is a major outlet for water coming from the Rhine. To protect Rotterdam from flooding a storm surge barrier in the Nieuwe Waterweg was constructed: the *Maeslant Kering*. An impression of the barrier is given in figure 1.

The requirements that Rotterdam should be protected from flooding, that its port should be reachable at all times (except at unacceptable weather conditions), and that the water coming from the Rhine should not cause Rotterdam to be flooded from the inside, has led to the design of a movable barrier. The barrier consists of two hollow floating walls, called

sector doors, connected with steel arms to pivot points on both banks. The construction, which should resist the huge forces of the incoming water, are as large as the Eiffel Tower. During normal weather conditions the two sector doors rest in their docks. Only when storms are expected with danger of flooding the two sector doors are closed. The closing procedure consists of several steps. First the docks are filled with water, which makes the doors float. Then the doors are moved to the centre of the Nieuwe Waterweg and then they are filled with water until they touch the bottom. A big advantage of the design of the movable barrier is that the construction and maintenance can be done without interfering with the ship traffic.

The main requirement on the barrier is that it must be as reliable as a dike. Careful failure analysis showed that manual control of this barrier would undermine the reliability. A normal human being has a failure probability of one in thousand for a complex task like deciding when to close the barrier and then closing it. Therefore it is safer to let a computer control the barrier.

The BOS System The control system which decides about opening and closing of the barrier and which also completely autonomously performs these tasks, was baptized Bos (Dutch: *Beslis & Ondersteunend Systeem*, i.e., decision and support system). When calculated predictions indicate that the expected water level in Rotterdam will be too high, Bos has the responsibility to close the barrier. Since Rotterdam is a major port with a lot of ship traffic, the barrier should be closed only when really necessary and as shortly as possible. An unnecessarily closed barrier will cost millions of guilders because of restricted ship traffic, while there is also the danger of flooding through the Rhine if its water cannot flow freely to the sea.

The design of the Bos system is an effort linking several distinct disciplines, viz., the organizational and global overview of the system functionality and requirements by *Rijkswaterstaat* (RWS), the hydrological knowledge and model-based water level predictions by the *Waterloopkundig Laboratorium* (independent research institute for water management and control), and the controlling and automation discipline and systems' integration knowledge by CMG.

Building a Safety Critical System Because of the dangers and costs involved, very strict safety and reliability requirements are imposed on the Bos software. The failure probability for not closing the barrier when this is deemed necessary should be less than 10^{-4}, and the failure probability for not opening the barrier when requested should be less than 10^{-5}. The latter is more critical because of the danger of destruction of the whole barrier if, due to water flowing from the Rhine, the pressure at the inside, i.e., land-side, of the barrier is higher than the pressure from the sea-side.

The high safety and reliability requirements make that the Bos is a *mission critical system* (or safety critical system), for which special care, effort and precautions should be taken in order to guarantee its safe, reliable and correct operation. To this extent, the design and development of the Bos software were guided by the standard IEC1508 [IEC], which is applicable to software development for safety critical systems. It is a best practices standard which categorizes systems according to their safety and reliability requirements into different *Safety Integrity Levels* (SIL). According to this categorization Bos belongs to the highest SIL level (SIL 4). IEC1508 denotes methodologies, techniques and activities as "not recommended", "recommended", "highly recommended", etc. depending on SIL level. For SIL 4 inspection and reviewing, use of an independent test team and the use of formal

methods are "highly recommended".

None of the "highly recommended" techniques in IEC1508 can assure completely the required safety, reliability and correctness. Only a carefully chosen combination of appropriate techniques can help to increase the confidence that the system has the required quality. This has led to the formulation of an integral validation and verification approach for the Bos system, which is documented in the Validation & Verification Plan (V&V-plan). The V&V-plan describes the scope and interaction of all the activities for quality assessment. Traditional testing is still an important part of quality assessment, now it is increased in effectivity and efficiency by the integration with other quality assessment and quality improvement techniques, such as the use of formal methods modelling, simulation and validation, reviews and inspections, static code checking, coding and documentation standards, developer testing (glass-box testing or debugging), module testing, integration testing and system testing.

Formal methods, being one of these techniques, is the focus point of this paper. With formal methods, systems are described as mathematical models. Due to their mathematical underpinning these models allow for precise specification and design description, formal (automatic) verification of system behaviour, simulation, derivation and calculation of system properties, and derivation of test cases.

In the development of Bos the formal methods Promela and Z were used for specification of the design. Promela is a formal language for modelling communication protocols, which is based on automata theory. It is the input language for the model checker Spin [Hol91, Hol97, Spi]. Z is a formal language based on set theory and predicate logic [Spi92].

3 Seven Myths of Formal Methods

The *Seven Myths of Formal Methods* presented in [Hal90] are seven claims about formal methods and their use, both positive and negative. Also in [Hal90] these claims are discussed and challenged, and, where applicable, counter-arguments against these claims are put forward. This section discusses the use of formal methods in the Bos project by considering to what extent these seven myths are true for the Bos experience.

Myth 1: *Formal methods can guarantee that software is perfect*

Some people, though not many and sometimes only for publicity reasons, claim that formal methods can guarantee correct software and that no other method can. It will hardly need argumentation to refute this claim: there is not a single method which can achieve perfection. Apart from simply being not true, it is a dangerous claim, because it sets high expectations on formal methods and it presupposes an all-or-nothing attitude towards formal methods. It would neglect that formal methods can be applied with different levels of formality, and thus lead to different levels of confidence in established correctness. In the Bos project formal methods were not applied with all their rigour. Their use during the development certainly helped in improving the quality of the system, but they do not guarantee correctness. A few remarks can be made with respect to not guaranteeing correctness.

The first remark concerns the fundamental problem of what correctness is. Software can only be proved to be correct if correctness is also stated formally, i.e., the specification or the user requirements should be formalized. But there is no way to formally prove that the specification is correct, i.e., that it is complete (it expresses everything) and that it is

valid (it expresses what was intended). Somewhere the link to the informal reality has to be made, and the validity of this link can only be assumed, not proved. Consequently, any formal proof only holds as far as the validity of the model on which it is based.

For the development of Bos the starting point was the *functional specification* (FS). The FS was developed on behalf of RWS and was input to the project. It was written in a combination of the Dutch language and classical structured design techniques such as Ward & Mellor and Hatley and Pirbhai [WM85, HP87]. It was decided not to formalize the FS but to start formalization at the level of the *technical design* (TD) which was developed on the basis of the FS. The control structure in the TD was expressed in the language Promela; its data structures and operations were expressed in Z. The formal TD had to be checked with respect to the informal FS. This was done by reviewing the TD, by having RWS review the TD, by simulation of the Promela descriptions using the tool Spin, and by informal argument on the Z descriptions. Many errors, ambiguities, inconsistencies and incompletenesses were found in the informal FS during this formalization process, many of which would very likely not have been found without formalization. But proof of formal correctness of the TD can never be obtained; only confidence in the correctness can increase.

Some aspects of the technical design were approached more formally. Model checking, i.e., a formal way of checking the validity of a property by going through the complete state space of a system model, was applied to the internal process scheduling mechanism and to most of the communication protocols between Bos and the outside world. For this, models were developed in Promela and subsequently simulated and verified using the model checking tool Spin. To make these verifications with Spin feasible, the models needed to be as simple as possible, which implied that the specifications themselves could not be used. Special models were developed for this purpose in which many abstractions had to be made. Again, validity of these models is an assumption and cannot be proved. The models were simulated and checked for a restricted set of generic properties such as the absence of deadlock and lifelock. This helped in gaining confidence that these protocols were functioning as expected.

The second point with respect to not guaranteeing correctness has to do with the restrictions of formal models. There is no single formalism which can express all aspects of system behaviour. In Bos only functional behaviour, as can be expressed by Promela and Z, was modelled; no real-time, performance, etc. aspects were formally described. Since these aspects are also important for Bos other methods of (informal) reasoning had to be used to argue for correctness for these aspects of behaviour.

The third reason that complete correctness cannot be guaranteed is simply by the fact that not the complete Bos system was formally specified. Some parts, such as the graphical user interface, were developed using conventional techniques. This is a usual case for using formal methods: parts of the system which are not considered to be critical, or for which there are other good methods of designing them, are not formally modelled.

The fourth, important reason that correctness cannot be guaranteed for Bos is that after development of the TD, the implementation and testing phases were not carried out formally. The formal specifications of the design were the basis for implementation and for testing, however, no formal derivation or verification of code, nor formal derivation of test suites were performed.

Development of code was carried out systematically and in a structured way based on the formal specifications. This process was guided by a kind of informally stated rules how to map a line of Z specification to statements in the programming language C++. This way of developing implementations turned out to straightforward, effective and efficient, although not formal. The quality of the code was relatively high, as could be derived from the small

number of implementation errors found during testing.

Analogous methods were used for the derivation of test suites. The testing process was conducted completely as without the use of formal methods, the only difference being that tests were derived systematically, though informally and manually, from the requirements in the formal specifications. Also the testing process turned out to be well-structured, effective, efficient and with a very high code coverage. More on testing in Bos can be found in [GWT98].

The main reason for not carrying out the implementation and testing phases in a formal way is that this is beyond the state of the art (and beyond the state of the tools) for a system of the size and complexity of Bos. The formal specifications for Bos consist of more than 20,000 loz (Lines Of Z). Neither manually, nor with any existing tool, we thought it to be feasible to operate completely formally on a specification of this size. Some trials with tools confirmed our thoughts in this respect. Also for testing there are currently no methods or tools that can derive tests formally and automatically from such specifications, although it is expected that developments in the testing area will proceed faster than in the implementation area, so that tools for automatic formal test derivation might be available within the next couple of years [PS97, VT98, BFV$^+$99, Tre99].

Another reason for not considering formal implementation development and formal testing was that Bos was the first project in which formal methods were applied on a large scale by CMG. It was felt that it would not be wise to use all the rigour and possibilities of formal methods in such a first project, but that it would be better to restrict the level of formality. Also in retrospective, this was considered as a good decision: on the one hand, the restricted use of formal methods already brought large benefits, while, on the other hand, introducing more formality would most likely have been too much for the already steep learning curve within this project. In a next, comparable project, CMG will certainly use the experience obtained in the Bos project and increase the level of formality.

Myth 2: *Formal methods are all about program proving*

No program was proved for Bos. As explained above, the application of formal methods in Bos was mainly restricted to the formal specification at design level (TD). Moreover, some model-checking and simulation were performed using Promela models and some hand-waving argumentation was used mainly to be convinced of the completeness of Z operation schemas. Program proving was not a main goal in the beginning and, when looking back, many observations support this view. Most likely, also in a next, comparable project CMG will not concentrate on program proving, even if in such a project the level of formality will be higher.

The first observation is that already the process of formal development of the TD helped a lot in finding omissions, ambiguities, inconsistencies and incompletenesses in the informal descriptions. The formalization process forces to think thoroughly, to look in more detail, to be precise, and to denote aspects unambiguously. Moreover, it allows to argue about the system in a better structured way with better mutual understanding between designers, and it makes it much more difficult to hide or obscure problems in vague phrasing. Consequently, the TD is more detailed than it had been without the use of formal methods. If no formal methods had been used, many of the problems and errors found would, most likely, only have been found during testing or, even worse, during operation. It is well-known that errors detected later in the development process are much more expensive to repair than errors found early during the development. This early detection of problems and errors is one of

the major benefits of the use of formal methods. In itself, it already justifies the use of formal methods, even without doing any formal program proving.

The second observation is that the formal description of the design constituted a precise, complete and unambiguous basis for implementation and for testing. Although the testing and implementation phases were not carried out in a formal way, the formal TD was of big help in making these phases effective and efficient, as described above. Moreover, the formal TD served as a precise arbiter for the resolution of differences in interpretation between implementers and testers. More formality in the implementation and testing phases, in particular adding program proving, could be nice but certainly does not have high priority within Bos. Given the effectiveness, efficiency and the relatively small number of problems occurring in these phases, the general feeling is that extra formality would not create much extra profit. This holds even more when considering the necessary extra investments, given the state of the art in program proving for large and complex systems like Bos, and when comparing with the expected profit which could be obtained in the more problematic, early phases of development.

The experience of developing Bos is that the early phases of development are the most critical ones. Important problems are encountered during capturing of requirements, understanding the problem, understanding the client, system specification, design, and checking that the design (and not the program) meets the requirements and solves the problem. In these phases the use of formal methods has most benefits, but, despite the currently achieved benefits, the impression is that the problems encountered in these phases are still larger than in implementation and in testing, and that, consequently, much more can be gained.

However, it is important to note that experiences in Bos showed that formal methods are not an ideal solution for some of the problems encountered in the early phases. In particular, this concerns problems of requirements capturing, understanding the problem, high level structuring of descriptions and conceptual modelling. Informal notations like those from Hatley & Pirbhay [HP87] were used together with formal methods, but the integration of the two was certainly not optimal. This should be improved or replaced by other techniques which allow better modelling and structuring at a conceptual level.

The consequence is that, for the Bos team, program proving and the formal and (semi) automatic derivation of code from formal specifications does not have high (research) priority. Analysis of encountered problems from Bos supports this view by showing that there were not many errors introduced during the implementation process. This observation is in conflict with current research on formal methods where a lot of effort is devoted to formal program (code) derivation and program verification.

Myth 3: *Formal methods are only useful for safety-critical systems*

Certainly, the Bos system is safety-critical and, certainly, formal methods were useful. From this, of course, it cannot be concluded that formal methods are not useful for other kinds of systems. First, note that nowadays almost any serious software system is in some sense critical, be it for safety with respect to humans, for material damage it may cause, for economic loss if an error is replicated in thousand copies of the software, or for the company image and profile.

A major reason for using formal methods in less-critical systems would be that the Bos project shows strong evidence that a software product of better quality can be made for the same price. One remark should be made with respect to the price: quite some effort has

to be spent on learning how to deal with formal methods (see below). This implies that that for a one-time, non-critical software development project, with a formally untrained and unexperienced development team, the use of formal methods might not be profitable.

Myth 4: *Formal methods require highly trained mathematicians*

Formal languages are based on high-school mathematics and logic. The Bos experience is that any educated software engineer can easily learn a language like Promela and, with just a bit more effort, a language like Z. You certainly need not be a highly trained mathematician to learn a formal language, on the contrary, highly trained mathematicians can be a burden for a formal methods project. Highly trained mathematicians may be too perfect in applying the formal techniques, aiming at mathematical elegance of expression instead of at practical and workable solutions. As an example, we have had mathematicians involved who were searching for the shortest expressions in Z to express particular solutions, but these solutions were so short and tricky that nobody else, in particular no implementer or tester, could read them. Such solutions do not bring the application of formal methods any further.

However, there is a point in learning formal methods. The Bos project made us aware that learning a formal *language* does not imply that a formal *method* can be effectively applied. We discuss a number of aspects of the learning trajectory, which have not so much to do with mathematics.

One point is the problem of making models. This involves choosing the right abstraction level and deciding what to model and what not. This is an intrinsic problem, not so much dependent on formal methods. It turns out that people having experience in making abstract models, whether in software engineering or not, have an advantage. How to make a good model at the right level of abstraction is very difficult to teach and must be learned by experience. This takes time and effort. The first models and specifications developed in the Bos project were certainly not the best ones.

A second point, which particularly applies to Z, is the large expressivity. Z is very expressive with respect to software engineering; it comprises typed set theory with equality and predicate logic. This implies that there are numerous ways to write the same thing. In order to be useful within a project, a specification style restricting Z, providing pragmatics, and giving structure to specifications, has to be found and adhered to, so that analogous problems get analogous solutions. But, for 20,000 loz (lines of Z), this style should also include simple things like lay-out and naming conventions, i.e., *specification standards*, analogous to coding standards which are very common in, e.g., Cobol programming. Without such conventions writing a Z schema each time starts from scratch, and more importantly, reading and understanding a Z schema each time has to start with figuring out the structure of the schema. If specifications are not written using the same style it is almost impossible for implementers and testers to read them efficiently and effectively. It took quite some time to discover that such a specification standard was necessary, to develop one and to impose it upon the specifiers.

A third point of learning concerns the combination of formal methods with other software engineering techniques and current software engineering practice. Formal methods do not solve all software engineering problems, so other methods and techniques were used such as version and configuration management, conceptual modelling techniques (Ward & Mellor, Hatley & Pirbhai, see above), reviewing and inspections, software planning and control, problem tracking, different kinds of testing (developer/white box, functional/black box, duration test) and quality assurance. The interactions of all these aspects with formal

methods were new, and had to be learned, mostly by experience.

One particular point of learning by experience was how to combine the formalism for dynamic behaviour specification PROMELA with the formalism for data- en operation specification Z, i.e., it had to be learned, mostly by experience, what to specify in which formalism.

A difficult point about which there is still no complete agreement among the BOS formalists, is how formal methods can best be taught. Within BOS they were taught using a combination of (short) courses and on-the-job training with supervision of a formal methods coach. The courses were considered to be too short and too theoretical and many engineers did not have the feeling of mastering the formal technique. Moreover, when the courses were given, the specification standards had not been discovered and devised, which meant that the full expressivity of Z was unnecessarily taught during the courses. On-the-job training helped but implied that quite soon after the course people were working on real products, with all possible consequences when making errors. A lot of engineers felt that they did not have enough opportunities to make errors without serious project consequences, and it is especially by making errors that you learn the most. On the other hand, when working on case studies not related to the project, there is a risk that these case studies are not performed as seriously because they may be just considered for playing, while, from the project management view, non-productive, precious time would be lost. The same may apply if learning formal methods is completely decoupled from any project. In the beginning it was difficult to find a coach with sufficient knowledge of formal methods and with sufficient experience in software engineering with a BOS-like project. The Formal Methods & Tools group of the University of Twente, supporting CMG with the use of formal methods, did not have the latter experience, while in the beginning CMG did not have knowledge about formal methods. Also the question whether the coach should be involved in the project and have project responsibilities, or should be external, has not been unanimously answered.

A final remark concerns the level of formality with which formal methods are applied. Most of the remarks above apply to writing and reading formal specifications. If more is required, in particular if formal proofs of properties or of correctness of code are required, then, of course, high-school mathematics are not sufficient. This is especially true since the tool support for such proofs is minimal. But not everybody is required to reach this level of formality; a few proof specialists within a project will usually suffice. Also in BOS there were only a few persons doing formal model checking.

Myth 5: *Formal methods increase the cost of development*

The feeling within BOS is that the costs of development with formal methods were comparable to, perhaps even a bit higher than without the use of formal methods. The main benefit of their use was in increased quality of the final product, not in reduced costs. However, a few remarks should be made with respect to these observations.

Formal methods were used in the critical path of the fixed price/fixed time BOS project. This means that there was no BOS developed without the use of formal methods, so a real comparison between with and without formal methods cannot be made. The observations above are merely based on experience of software engineers obtained in previous projects.

A second remark concerns the long learning curve, which was part of the project. Learning included both learning in courses as well as making many mistakes when applying the methods for the first time. It is expected that a next BOS-like project with the same team of software engineers will save time and money.

As a third remark it is noted that, independent of decrease or increase of costs, there was a major shift in costs and efforts over the phases of the development trajectory. The specification and design phases of the project were much more expensive and took much more time. This implies that much more of the project's budget had been consumed before the first line of programming code appeared. Managers not prepared for this shift, will probably get very nervous when this happens. Moreover, managing these long specification and design phases and observing progress in these phases is difficult. There are no established methods yet to measure progress of formal development. Fortunately, the additional efforts and costs spent in the first phases really mean that more work has been done and that the implementation and testing phases are proportionally shorter.

Finally, it should be observed that costs are, of course, different if complete formal proofs or formal verification are required. Then the total costs are likely to be much be higher, but the product quality is also expected to improve.

Myth 6: *Formal methods are unacceptable to users*

The user of Bos, RWS (Dutch Ministry Transport, Public Works and Water Management) was satisfied with the formal designs and with the final product. RWS looked at all intermediate formal designs, although not at every detail. The formal descriptions were accompanied by informal explanations and comments, which is certainly necessary. By using formal descriptions it was easier for the designers to point out and explain errors, ambiguities, inconsistencies and unclarities in the informal descriptions.

A very useful way of pointing out and explaining defects was to use the simulator functionality of the tool Spin. Scenarios of sequences of events leading to problems were simulated and presented using the very intuitive Message Sequence Chart (MSC) notation. One of the successes (at least from the formal methods point of view) was when it could be shown, using a Promela model and the tool Spin, that one of the important protocols for communication between Bos and the outside world, developed by a third party, had a serious problem. This protocol allowed behaviour resulting in the unsafe situation that one of the sector doors would be closed while the other would remain opened. Using the MSC functionality of Spin the sequence of events leading to this unsafe situation could be easily demonstrated to managers and to designers of the protocol, who were immediately convinced of the problem. Without the use of formal methods and without the easy way of demonstrating using MSC, it would have been much more difficult to explain the error and to convince them that corrective action was necessary.

Myth 7: *Formal methods are not used on real, large-scale software*

In the Bos project formal methods were used successfully. The success of their use is probably best illustrated with the answer that the software engineers gave to the question how they would approach a next, Bos-like project. The answer was unanimously that they would use formal methods, while some of them, who had no previous large-scale software engineering experience, wondered how people had ever managed to do without formal methods.

4 Concluding Remarks

The main conclusion is that the use of formal methods helped improving the quality of the Bos system. Since this was the main goal of the use of formal methods, it can be concluded that they were applied successfully. This does not imply that there were no problems encountered: learning to apply formal methods and the integration with existing software engineering methods were some of the weak points identified in section 3, while also the level of formality could be increased in a next, analogous software development project.

When considering the seven myths of [Hal90], we see that the Bos experience mostly agrees with the observations made about these myths in [Hal90]. Some minor differences can be observed between the Bos experience and [Hal90]; these are briefly discussed.

The first observation is that in Bos there is a clear distinction between the specification (functional specification – FS) and the design (technical design – TD). The FS was externally given and not formalized, while the TD was formally developed. Retrospectively, when considering the number of problems, errors, ambiguities and inconsistencies found in the FS while developing the TD, it would probably have been better if also a formal FS had been developed. More rigorous checks of the TD with respect to the FS would have been possible in that case.

The explicit distinction between specification and design is not made in [Hal90]. A formal system specification is assumed from which an implementation is derived in one or more, formal or informal, steps. Usually, in software development, a separate design phase is recognized. When this is the case, as in Bos, the question is what to formalize: the specification, the design, the transition from specification to design, from design to implementation, or a combination of these.

The second observation concerns the difference between a formal specification and a formal model. In [Hal90] they are not distinguished, but the experience in Bos shows that they are different things, which are both necessary. Specifications are meant to specify the system, i.e., prescribe their behaviour, in which they should be complete. A specification is usually too complex and contains too much detail for checking of properties, e.g., model checking. This is especially true if property checking is performed using a tool; state of the art tools cannot deal with large and complex formal descriptions. Hence, for checking of properties a formal model is developed. This model is an abstraction of the system in which a lot of detail has been removed and only those aspects which are deemed important for the property at hand, are formally expressed. A model will usually be simpler than the specification so that it can be handled by tools. The model, on the other hand, cannot be used as a specification since it is not complete. Making models is an intricate process in which the right level of abstraction must be chosen to achieve a good compromise between simplicity and completeness.

The third difference concerns conceptual modelling. The experience of Bos is that formal methods are not ideally suited for making conceptual models and high level structuring of specifications and designs. This contradicts [Hal90]. In Bos structured development techniques, like Ward & Mellor and Hatley & Pirbhai were used. Probably, object oriented modelling techniques could be used well.

A last point, which is almost completely neglected in [Hal90] – but which is considered in *Seven More Myths of Formal Methods* [BH95] – is the use of formal methods tools. In Bos, tools were considered to be very important in different respects. Tools help a lot in learning a formal technique. Of course, this includes static checking of specifications, but even more

important, tools stimulated people to work with formal methods and challenged them to do experiments with them. In this respect the tool SPIN was very successful: everybody can get the tool from the Web [Spi], play with it, write PROMELA models or use the example models, simulate them, and see the results. This kind of experimenting with a tool is instructive, gives fun, and stimulates the use of formal methods. For Z there were only static tools available (the static semantics checker ZTC) and this does not stimulate as much the use of formal methods. Also during the development of the formal TD of BOS tools helped in stimulating the use of PROMELA by allowing that "something could be done" with the formal descriptions. In this respect, engineers found the use of Z sometimes frustrating, because "you couldn't do anything with the specifications".

During the actual development of the formal TD the tools which were used, SPIN and ZTC, were useful. It helps a lot if some checking can be performed on specifications consisting of several thousand lines of Z, such as checking of type consistency, declaration of variables, etc. A lot of simple mistakes with respect to the language definition could be detected in this way. With PROMELA and SPIN some validations were performed and in this respect the usage of tools was also successful, see section 3.

In general, however, tools support for formal methods was found to be insufficient. This applies both to the functionality of the tools and to the size of specifications which can be handled. Although SPIN is one of the most efficient model checkers available, a large effort had to be put in making the models fit into SPIN. For Z, the only workable tool support was static checking. Dynamic tools (theorem provers; especially precondition checking of Z operation schemas was desired) have difficulties handling the size of the specifications occurring in BOS and if they fit the effectively usable functionality is restricted. Moreover, the integration of formal methods tools with other software engineering tools, which is an important issue for working efficiently in large projects, was very immature, i.e., non-existing.

Acknowledgements

Eric Burgers, Wouter Geurts, Franc Buve, Rijn Buve, Sjaak de Graaf, Hedde van de Lugt, Peter Bosman, Peter van de Heuvel and Robin Rijkers are acknowledged for their enthusiastic collaboration during the interviews on which this paper is based. While performing the work reflected in this paper the first author was financially supported by CMG The Netherlands. René de Vries and Jan Feenstra are thanked for proof-reading.

References

[BFV+99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*. Kluwer Academic Publishers, 1999.

[BH95] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[CTW99] M. Chaudron, J. Tretmans, and K. Wijbrans. Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System. In *FM'99 – World Congress on Formal Methods in the Development of Computing Systems*. Lecture Notes in Computer Science, Springer-Verlag, 1999. To appear.

[GWT98] W. Geurts, K. Wijbrans, and J. Tretmans. Testing and formal methods — Bos project case study. In *EuroSTAR'98: 6^{th} European Int. Conference on Software Testing, Analysis & Review*, pages 215–229, Munich, Germany, November 30 – December 1 1998.

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, 6(9):11–19, 1990.

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

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HP87] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, 1987.

[IEC] IEC. *Functional Safety: Safety Related Systems*. International Standard IEC 1508.

[Kar97] P. Kars. The Application of Promela and Spin in the Bos Project. In J.-C. Grégoire, G.J. Holzmann, and D. Peled, editors, *The Spin Verification System: The Second Workshop on the Spin Verification System; Proceedings of a Dimacs Workshop, August 5, 1996*, volume 32 of Dimacs *Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–63. American Mathematical Society, 1997.

[Kar98] P. Kars. Formal Methods in the Design of a Storm Surge Barrier Control System. In G. Rozenberg and F.W. Vaandrager, editors, *Lectures on Embedded Systems*, pages 353–367. Lecture Notes in Computer Science 1494, Springer-Verlag, 1998.

[PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.

[Spi] Spin. On-the-Fly, LTL Model Checking with Spin.
URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[Spi92] J.M. Spivey. *The Z Notation: a Reference Manual (2nd edition)*. Prentice Hall, 1992.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99*. Lecture Notes in Computer Science, Springer-Verlag, 1999.

[VT98] R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using Spin. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2, 1998. Ecole Nationale Supérieure des Télécommunications. Also to appear in *Software Tools for Technology Transfer*.

[WB98] K.C.J. Wijbrans and R. Buve. Software bestuurt de stormvloedkering. *Software Release Magazine*, 50(5), 1998. In Dutch.

[WBG98] K.C.J. Wijbrans, F. Buve, and W. Geurts. Practical Experiences in the Bos Project. In *Proceedings of the Embedded Systems Symposium*, Eindhoven, The Netherlands, May 19, 1998. Eindhoven Univerity of Technology.

[WM85] P.T. Ward and S.J. Mellor. *Strucured Development for Real-Time Systems*. Yourdon Press Computing Series. Prentice-Hall, Inc., 1985. Volume 1: Introduction & Tools.