Formal Methods in System Design, 19, 195–215, 2001

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

JAN TRETMANS tretmans@cs.utwente.nl
University of Twente, Department of Computer Science, Formal Methods and Tools Research Group, P.O. Box 217, 7500 AE Enschede, The Netherlands

KLAAS WIJBRANS* klaas.wijbrans@cmg.nl
MICHEL CHAUDRON† m.r.v.chaudron@tue.nl
CMG Public Sector B.V., Division Advanced Technology, P.O. Box 187, 2501 CD The Hague, The Netherlands

Received November 30, 1999; Accepted March 15, 2000

Abstract. This paper discusses the use of formal methods in the development of the control system for the *Maeslant Kering*. The *Maeslant Kering* is the movable dam which has to protect Rotterdam from floodings while, at (almost) the same time, not restricting ship traffic to the port of Rotterdam. The control system, called BOS, completely autonomously decides about closing and opening of the barrier and, when necessary, also performs these tasks without human intervention. BOS is a safety-critical software system of the highest Safety Integrity Level according to IEC 61508. One of the reliability increasing techniques used during its development is *formal methods*. This paper reports experiences obtained from using formal methods in the development of BOS. These experiences are presented in the context of Hall's famous "Seven Myths of Formal Methods".

Keywords: industrial application of formal methods

1. Introduction

BOS is the software system which controls and operates the storm surge barrier in the Nieuwe Waterweg near Rotterdam (figure 1). 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 have no clear advantages and are unworkable. Some of these claims have almost become myths. A number of these myths are described

*Current address: CMG Wireless Data Solutions (Netherlands) B.V., P.O. Box 1893, 6201 BW Maastricht, The Netherlands.
†Current affiliation: Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

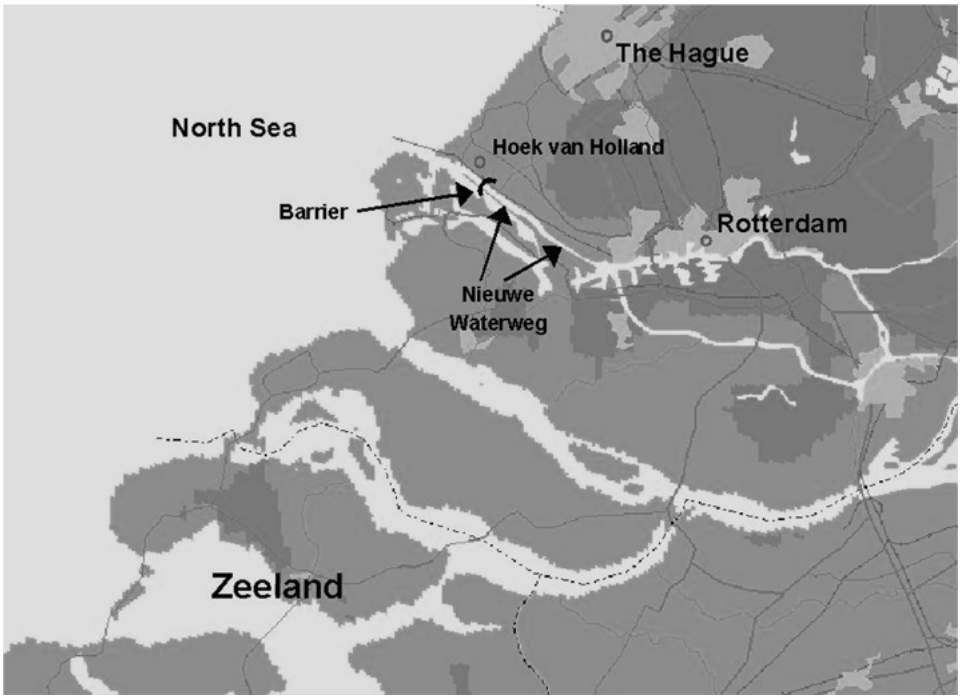


Figure 1. The geographical situation of the *Maeslant Kering* near Hoek van Holland, between Rotterdam and the North Sea.

and discussed in a famous article: *Seven Myths of Formal Methods* [8]. The experiences obtained from using formal methods for the development of BOS will be discussed on the basis of that article. We will discuss to what extent those myths are true for the BOS project.

The data for this paper 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.

This paper concentrates on the experiences of these software engineers with formal methods. These experiences are reported in Section 3 with the "seven myths" as guiding and structuring principle. This paper does not attempt to give a full and systematic analysis of all aspects involved in using formal methods, nor does it discuss technical details about the particular formal methods used or the way they were used (these were described in [14, 15]). Moreover, formal methods were only one of the techniques 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 only briefly described in Section 2.3; more elaborate accounts can be found in [22, 23], while testing of BOS is described in more detail in [7]. This paper elaborates [19], while taking also some elements from [5].

The core of this paper is presented in Section 3 where the seven myths are challenged with the BOS experience. Before that, Section 2 describes the BOS system, the reason for its development, its development process, and some characteristics of the system. Section 4 gives some concluding remarks.

2. The BOS system

2.1. The storm surge barrier

The Netherlands are located in a low delta near the North Sea, into which important rivers such as the Rhine and IJssel flow. The history of The Netherlands 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 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 and 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. Also, 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 near Hoek van Holland: the *Maeslant Kering*. The geographic map is depicted in figure 1, while an impression of the barrier is given in figure 2.

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 land-side, 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. Each of these, which should resist the huge forces of the incoming water, is 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 hollow doors float. Then the doors are moved to the centre of the Nieuwe Waterweg and filled with water to make them sink. A big advantage of the design of this movable barrier is that the construction and maintenance can be done without interfering with the ship traffic. For more information, we refer to the internet-site of the Dutch Ministry of Transport, Public Works and Water Management where an animation of the moving barrier is provided, see [6].

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 was considered to be safer to let a computer control the barrier.

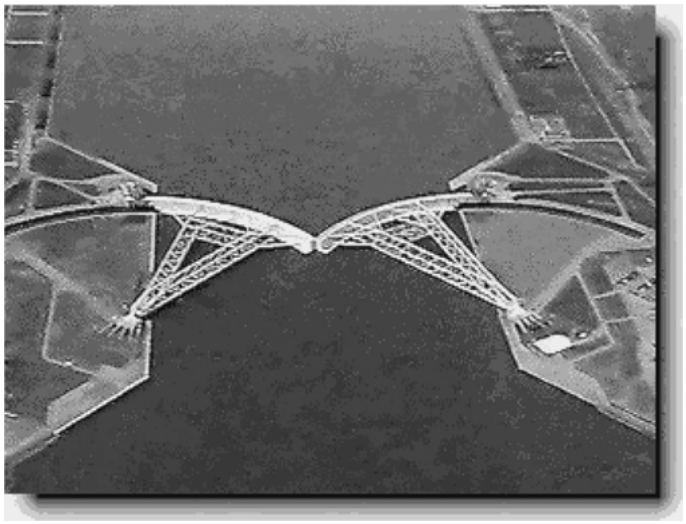


Figure 2. Top view of the *Maeslant Kering*. At the top of the figure the Nieuwe Waterweg flows to the North Sea; in the bottom direction is Rotterdam. The Nieuwe Waterweg is about 300 m wide.

2.2. *The* Bos *system*

The control system that decides about opening and closing of the barrier and that 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 briefly 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 when its water is blocked by the barrier.

The realization of the BOS system is an effort in 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.

2.3. *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.

These 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 IEC 61508 standard [12], which gives guidelines for software development for safety critical systems. IEC 61508 is a best practices standard that categorizes systems into different *Safety Integrity Levels* (SIL) according to their safety and reliability requirements. According to this categorization BOS belongs to the highest level: SIL 4. IEC 61508 denotes methodologies, techniques and activities as "not recommended", "recommended", "highly recommended", etc. depending on SIL level. Some of the "highly recommended" techniques for SIL 4 are inspection and reviewing, use of an independent test team and *the use of formal methods*.

None of the "highly recommended" techniques in IEC 61508 can by itself completely assure 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 a dedicated system engineering process for the project. This dedicated engineering process was ISO 9001 certified. Figure 3 indicates which techniques have been used in the different phases of development within this process.

An integral, risk oriented approach in the system development path identifies at an early stage the aspects of the system that are critical for carrying out the mission of the system. These risks are managed by carrying out both process and product measures. In

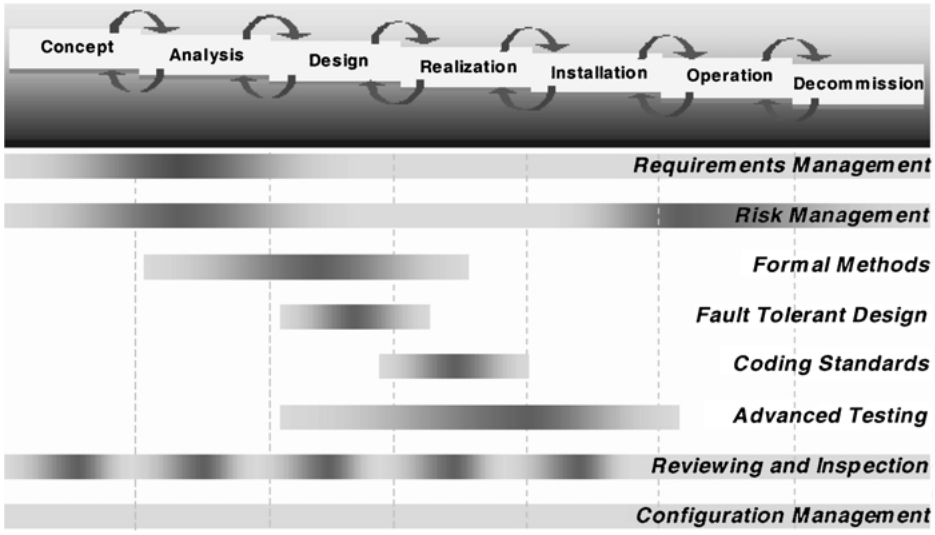


Figure 3. The dedicated system engineering process combines different techniques.

the development path, a number of methods and techniques aimed at quality improvement and quality assessment, are integrated. These are mutually supporting and bring about an effectiveness of the whole that is greater than the sum of the parts. So, there is traditional testing which is still very important, but 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. Moreover, the process is risk-oriented: each quality effort is adapted to the level of criticality of the part of the system considered.

2.4. Formal methods in the Bos development trajectory

One of the "highly recommended" techniques applied in the BOS system development is formal methods. With formal methods, systems are modelled using precise mathematical concepts. Due to their mathematical underpinning these models allow for precise specification and design description, formal (automatic) verification of system behaviour, simulation and animation, 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 modelling and specification of the design (see [5] for the rationale for the selection of these methods). 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 [10, 11, 17]. Z is a formal language based on set theory and predicate logic [18]. This subsection indicates where and how formal methods were applied in the different phases of the BOS development process.

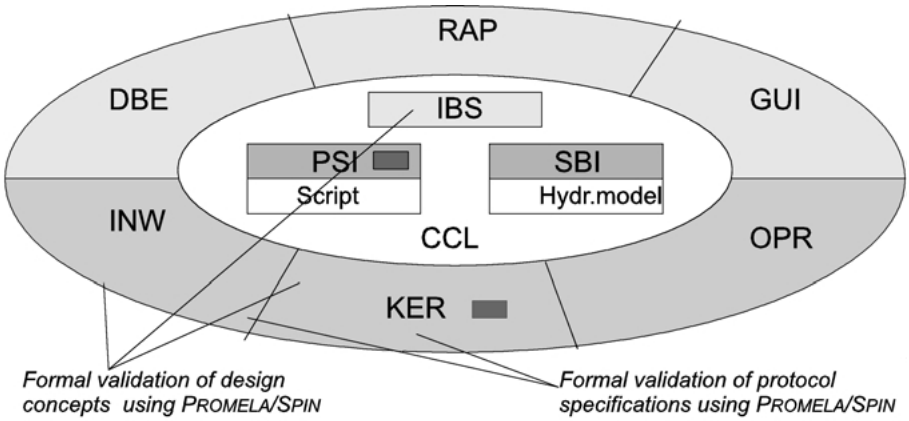
2.4.1. Specification. Starting point for the development of BOS was the *functional specification* (FS). The FS was developed by the customer—RWS—and was input to the project. The FS consisted of a natural language document enhanced with data flow diagrams with in total approximately 700 pages of natural language text. The FS described the desired functionality of the BOS system. This functionality was already decomposed into a set of functions based on the data flow diagrams. The FS was not a pure specification of the system; it was mixed with implementation decisions and algorithm descriptions. For example, the handling of the communication protocol for several of the specific interfaces was described in detail.

The main formal activity that was performed on the FS was the analysis of the communication protocols between BOS and its environment. These communication protocols were modelled in PROMELA and validated using the SPIN tool. The validation showed that the proposed protocol handling contained fatal flaws. Alternative protocol specifications were developed and validated with SPIN. These alternatives were proposed to RWS and accepted.

Other kinds of thorough analysis resulted in several proposals from the project team to the customer for changes to the specification that would improve the reliability of the system. These were all accepted.

2.4.2. Architecture. The main objective of the *Maeslant Kering* is the protection of Rotterdam from flooding. Based on this objective a *function failure analysis* (FFA) was made. Based on the FFA the criticality of the whole barrier, of BOS, and of each subfunction with respect to performing the barrier operation was determined. The overall outcome of this analysis is that the 'safety' of Rotterdam mainly depends on the 'reliability' of the BOS system. Based on the function failure analysis, an architecture that maximizes decoupling of critical and non-critical parts was developed. This resulted in the architecture depicted in figure 4, which shows the different subsystems of BOS. The three-letter names of the subsystems are Dutch abbreviations.

Part of the architectural effort was the creation of several patterns that were reused in the different subsystems. The most important are a generalized approach to recovery within



Core:	
PSI	Script Interpreter that executes the decision script of Rijkswaterstaat.
SBI	Interface to the hydraulic model.
CCL	Communications and utilities library.
IBS	Starting, stopping, monitoring and controlling the other subsystems and error handling and recovery.
Critical:	
INW	Obtaining measurement data from all kinds of weather and water sensor networks.
KER	Control of the barrier PLC hardware itself.
OPR	Communication by fax, semaphone and direct terminal communications.
Noncritical:	
GUI	User interface of the Bos system.
RAP	Report generation after barrier operation.
DBE	Archiving data obtained during operation of the system and cleaning it up afterwards.

Figure 4. Bos subsystems with indication of the application of formal methods to them.

the BOS system (subsystem IBS), a generic approach to error handling that prevents the generation of large streams of error messages when specific faults occur, and a common pattern for implementing the different critical interfaces.

The algorithms designed for the critical interfaces KER and INW were formally validated using PROMELA. Also, the recovery mechanism as implemented in IBS was formally validated using PROMELA. Deadlock freedom was realized for the complete design by adhering to the design rules of [16].

2.4.3. Detailed design. Based on the approach described in Section 2.4.2, a detailed design was made. This was done using the Ward & Mellor method [21] in the CASE tool SDW [4]. From this CASE tool, the design documentation was generated using PERL scripting and LaTeX. It is common practice to use only natural language for specifying data elements, data flows and data stores, and to use pseudo code for specifying the operations. In contrast, the BOS project employed a combination of natural language and the formal language Z for the specification of both data and operations. On the Z specifications different activities were performed.

Firstly, the consistency of the design was statically checked. From the formal models in the database of the CASE tool, Z specification were extracted for single operations or for subsystems. Such Z specifications were then checked using the Z Type Checker ZTC [13] on the correctness of syntax, type correctness, and completeness of used and defined elements. This checking was done for all subsystems.

Secondly, reviewers examined and evaluated the pre- and post-conditions of the Z operation schemas, and compared them with the natural language text specifying the intention of the operation. This kind of reviewing by 'cross-referencing' between formal and natural language descriptions proved very valuable and led to the detection of many potential defects. This reviewing was done for all subsystems.

Thirdly, completeness of Z operation schemas was checked, i.e., for each schema it was checked that it covered any combination of current state and inputs. Checking was done by manual argumentation. It was done for INW, KER, OPR, DBE, PSI, and SBI.

Finally, for highly critical subsystems also the postconditions of operations and the invariants were checked. As this was a tedious manual process, it was not feasible to do it for all parts of the system. It was applied to parts of the PSI subsystem and to parts of the KER subsystem.

The final detailed design specifies 29 different programs. The generated design documentation consists of 4,000 pages of natural language text and Z. In total, 20,000 lines of Z were produced.

2.4.4. Implementation. In the implementation phase, the source code was created in a safe subset of C++. Development of code was carried out systematically and in a structured way based on the formal detailed design specifications. This process was guided by a kind of informally stated rules how to map a line of Z specification to statements in C++. No formal derivation of code nor formal verification were performed. Pre- and post-conditions and invariants from the Z schemas were implemented as C++ assertions in the code. Moreover, all subsystems used the same basic pattern that included all functionality for reporting errors

and recovering failed or hanging subsystem-processes. This way of developing implementations turned out to be straightforward, effective and efficient, although not formal.

The final implementation consists of 200 kLOC (kilo Lines Of Code) for the operational system and additionally 250 kLOC for simulators, test systems and supporting software.

2.4.5. Testing. Module and subsystem testing of BOS was done by an independent test team. This team based their tests solely on the formal specifications and was therefore not biased by the actual implementation of the software. The test cases were derived manually from the formal specifications. This derivation process, though informal, was very systematic, taking care that each precondition of each Z schema was covered and its postconditions checked. The approach turned out to be well-structured, effective, efficient, resulting in a test code coverage of 80–90% using pure black-box testing, which is relatively high. White-box testing was applied additionally to cover the remaining 10–20%. More on testing in BOS can be found in [7].

2.4.6. Some results. During the development of BOS, problem metrics were collected. In total, 1655 problems were submitted, 119 of these occurred during customer acceptance testing (FAT/SAT). Figure 5 shows the distribution of the problems over the project phases, where the phase indicates when the problem was detected. Moreover, figure 5 shows the location of residual faults found during acceptance testing by the customer. Relatively few faults were found during customer acceptance testing, and only three faults were found until now in operation (all of these three concerned the setting of configuration parameters). Most of the residual faults proved to be of cosmetic nature, in the user interface, in the debugger or trivial things—at least from a design and reliability point of view—like the

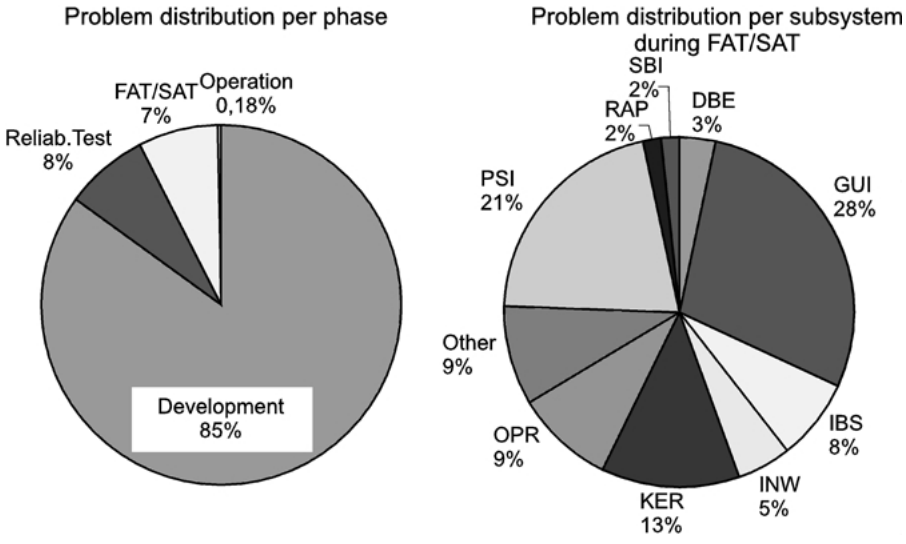


Figure 5. Distribution of faults per phase and break-down of FAT/SAT phase.

texts of error messages. Approximately half of the number of faults were code or design problems, the other half were incorrect test specifications, incorrect configuration parameters or documentation problems. None of the faults affected multiple subsystems in the design; all had limited scope and could be solved locally.

3. Seven Myths of Formal Methods

The *Seven Myths of Formal Methods* presented in [8] are seven (positive and negative) claims about the use of formal methods:

1. *formal methods can guarantee that software is perfect*;
2. *formal methods are all about program proving*;
3. *formal methods are only useful for safety-critical systems*;
4. *formal methods require highly trained mathematicians*;
5. *formal methods increase the cost of development*;
6. *formal methods are unacceptable to users*;
7. *formal methods are not used on real, large-scale software.*

Also in [8] 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 analysing to what extent these seven myths hold 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 also neglects that formal methods can be applied with different degrees of formality, and thus lead to different levels of confidence in established correctness. In the BOS project formal methods were not applied in the most rigorous possible way. Their use during the development certainly helped in improving the quality of the system, but they did not guarantee correctness. A few remarks can be made with respect to not guaranteeing correctness.

Starting point for formalization 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 of the user requirements should be formalized. This incurs two problems. Firstly, for any 'real world' system, it is practically impossible to capture all requirements. This is because a virtually infinite number of events may happen and interfere with a system in different ways. So, any specification will be an abstraction of the 'real world' system, potentially missing important aspects. Secondly, there is no way to formally

prove that the formalization of informal requirements 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.

The starting point for the development of BOS was the *functional specification* (FS), see Section 2.4. The FS was not formal. 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 SPIN tool, 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 these would very likely not have been found without formalization process. However, 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 (temporal) properties of the system by analysing 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 these checks, models were developed in PROMELA and subsequently simulated and verified using the SPIN model checking tool. 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 progress properties and the absence of deadlock and livelock. The improved understanding of the system and its environment and the better insight in the effects of design choices resulting from variations in the models proved more valuable than the bare 'yes' or 'no' answers to the checked properties. All this helped in gaining confidence that these protocols were functioning as intended.

Aspects of formalization The second point with respect to not guaranteeing correctness is related to the limitations of formal models. No single formalism can express all aspects of a system. In BOS only (untimed) behaviour, data and operations on these data were formalized; no real-time, performance or other non-functional aspects were described formally. Since these aspects are also important for BOS other methods of (informal) reasoning had to be used to argue for correctness of these aspects of the system (e.g., fixed time-slot scheduling, worst-case statistical performance analysis, etc.).

Even when restricting to the formalization of untimed behaviour, data and operations, no single formalism was found that combined these aspects in a suitable way and provided sufficient text-book and tool support. Consequently, two formal methods were selected for BOS: the behavioural view was modelled using PROMELA [10, 11] and the functional view

(of data and operations) was modelled using Z [18]. Hence, there was a need to relate the descriptions in the different formal languages to obtain a complete system model. This was done by using naming conventions in PROMELA and Z.

The use of multiple formalisms for describing one system brought along advantages as well as disadvantages. An advantage is that the system is considered from different points of view and that special attention is paid to connecting these views. The confrontation of the different views increases the likelihood of finding problems or omissions in an early stage. On the other hand, disadvantages are that the different specifications may overlap, thus introducing possible inconsistency and double work, or that the different specifications may leave certain systems parts unspecified, thus introducing incompleteness. Another disadvantage is that there is no tool support for the integrated use of PROMELA and Z. Although the use of one, integrated language would have been beneficial for the BOS project, the use of different formalisms was not considered to be a big hindrance by most developers.

Formalization of critical subsystems The third reason that complete correctness cannot be guaranteed is simply by the fact that it is not economically feasible to formally specify the complete BOS system. Some parts of the system, such as the graphical user interface, were developed using conventional techniques. This is a common approach in the use of formal methods: parts of the system which are not considered to be critical—based on failure analysis—or for which there are other good methods of designing, are not modelled formally.

Formalization in the development phases The fourth reason that correctness cannot be guaranteed for BOS is that formal methods were not used in all the phases of development; see Section 2.4. In particular, after development of the TD, the implementation phase was 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; see also MYTH 2.

Correctness of tools Finally, for systems with a complexity of the BOS system, it is not feasible to perform al formal analyses manually. Hence, tools have to be used. This raises the question whether the tools themselves are correct. Also, both manual proof and tool-assisted proof of complex specifications are themselves complex activities and thus error-prone. An incorrect proof will certainly give false confidence.

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)—simply because this promised the highest return on investment.

At the code level, formal methods where used through the inclusion of invariants and pre- and post-conditions from the Z specifications as C^{++} assertions. During debugging, normal testing and endurance testing these were found helpful and effective in finding a number of defects, however, this approach cannot be called formal program proving.

From the start of the project, the philosophy for the use of formal methods has been to use them for the verification of properties of the design rather than for proving correctness of programs. These approaches have two important differences. Firstly, there is the difference between focus on design versus focus on implementation. This is justified by the fact that it is more cost effective to remove problems and errors in an earlier development phase. Secondly, there is the difference between proving properties versus proving correctness. Properties of a system are more directly related to the externally visible behaviour, while correctness incurs a heavier proof-burden which is focussed on the (consistency of the) internal operation of a program.

In retrospect, many observations support the validity of focusing on properties of the design. In a next, comparable project CMG will most likely not concentrate on program proving, even if in such a project the level of formality will be higher.

The first observation is that the formalization process of the TD forces engineers to think (more) thoroughly, to look in more detail, to be precise, and to denote aspects unambiguously. The formal specification enables more structured reasoning about system properties, increases mutual understanding between designers, and makes it much more difficult to hide or obscure problems in vague phrasing. Consequently, the TD is more detailed than it would have been without the use of formal methods. If no formal methods would have been used, many of the problems and errors that have been found would, most likely, have only been found during testing or, even worse, during operation.

The second observation is that the detailed 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. 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 from a formal methods point of view but did certainly 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.

Moreover, increased formality in implementation and testing would be difficult to achieve given the state of the art, and given the formal methods' experience of the development team. The experience with large scale application of formal methods was such that is was decided not to use all the rigour and possibilities of formal methods in this project. The state of the art in program proving tools does not allow easy application to systems with the size and complexity of BOS, while also manual program proving is no option. Also for testing there are currently no 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–checking whether a system has a property is simpler than automatically generating a system which has that property [2, 20].

The general 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, and, despite the currently achieved benefits, the impression is that even much more can be gained here.

The experiences in BOS also 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 [9] were used together with formal methods, but the integration of the two was certainly not optimal. Typically, the modelling methods work well for describing programming abstractions, but fail to scale up to system level abstractions. This is indicative of the need for (formal) methods for modelling systems at an architectural level (as advocated in [1]).

The overall impression of the BOS team is therefore that program proving and the formal and (semi) automatic derivation of code from formal specifications should 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, for instance, if an error is replicated in thousands of copies of released software, or for the public image of a company.

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 for a one-time, non-critical software development project, with an unexperienced development team that is not trained in formal methods, their use may 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 use does not help to advance the application of formal methods.

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.

Making models One point is the problem of making models. This involves choosing the right abstraction level and deciding what to model and what not. This problem seems inherent to (software) engineering and is not in particular related to the use of formal methods. However, formal specification gives a false feeling of confidence, one is so busy writing the specification that one forgets about the much more important selection process of what to specify and what not to specify. It turns out that people who have experience in making abstract models, whether in software engineering or elsewhere, have an advantage (e.g., control engineers with experience in physical systems modelling). 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.

Degree of freedom of expression 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 the writing of a new Z schema starts from scratch each time anew. An even more important drawback of not using specification standards is that reading and understanding Z schemas starts with figuring out the structure of each individual schema. Conventions used in BOS range from the explicit statement of pre-conditions as the first line in operation schemas and the use of a constructive and deterministic style wherever possible to the use of certain lay-out conventions.

If all specifications are not written in 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.

Another difficulty of Z, related to the large expressivity, is that quite often a single typing error does not lead to a Z error, but to another, correct Z specification with a completely different meaning. The most prominent example of this—which actually occurred in the BOS project—is forgetting to write a single prime-symbol $'$ to indicate the new state of a state variable. This gives a completely different meaning to a Z schema. In this respect, Z is not very robust for small errors.

Combining formal methods with other techniques 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 [9, 21], 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 discovering how the PROMELA formalism for specification of dynamic behaviour and the Z formalism for specification of data and operations had to be combined. By experimenting, it was learnt what to specify in which formalism.

Teaching formal methods 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 formal methods 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 we did not 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 sufficient 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. Even then, domain knowledge about what to check is at least as important.

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 an empirical comparison between development 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.

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

Shift of development efforts An important observation is that, independent of decrease or increase of costs, there was a major shift in efforts and costs 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 program 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.

Moreover, the planning of the coding and testing phase based on formal specifications was improved considerably. It turned out that the number of lines of Z could be used as a rudimentary metric for planning. Analysis showed a correlation between the number of lines of Z and the number of lines of C^{++} code or the testing effort, respectively. In particular, after some experience had been obtained and some data had been collected, module test execution could be planned relatively precisely based on this metrics.

The effect of successful planning was strengthened by the fact that test plans and test suites were developed concurrently with coding. Although not directly influencing total costs, the overall development time was reduced by shortening the critical path for testing. BOS showed that the level of detail, preciseness and completeness provided by the formal design specifications makes it possible to perform most of the testing phase concurrently with code development. Starting from the formal design, implementers start coding, while at the same time testers start with writing test plans, developing an operational test environment, and making detailed specifications of test cases and implementing them. Usually, testers are faster so that when the implementation is ready, test execution can start immediately.

Moreover, testers were also heavily involved in the design specifications themselves, since they turned out to be very effective reviewers of design documents.

MYTH 6: *Formal methods are unacceptable to users*

The user of BOS, RWS (the Dutch Ministry of 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 open. Using the MSC functionality of SPIN the sequence of events leading to this unsafe situation could be easily demonstrated to designers of the protocol and even to managers, 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 by 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.

It should, however, be noted that formal methods are not an all ready, off-the-shelf technique for large practical applications. One has to be prepared to overcome difficulties and a significant effort has to be spent on tailoring in order to apply formal methods most effectively.

4. Concluding remarks

The main conclusion is that the use of formal methods helped to improve the quality of the BOS system. Since this was the main goal of the use of formal methods, it can be concluded that their application was a success. This does not imply that there were no problems encountered: learning to apply formal methods and integrating formal methods with existing software engineering methods and tools were some of the weak points identified in Section 3.

Also the level of formality could be increased in a next software development project with similar characteristics.

When considering the Seven Myths of Formal Methods of [8], we see that the BOS experience mostly agrees with the observations made about these myths in [8]. Some minor differences can be observed between the BOS experience and [8]; 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 [8]. 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 [8] they are not distinguished, but the experience in BOS shows that they are different uses of formal methods, which are both useful. Specifications are meant to specify a system as completely as possible, while models are meant to focus on one particular aspect of a system while abstracting from other aspects.

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. A model is a carefully chosen 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. Making variations of a model leads to better understanding and to more insight into the robustness for the property at hand. 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 [8]. In BOS structured development techniques, like [21] and [9] were used. In the early phases of system development there is a need for describing the system at a high level of abstraction. To adequately describe and analyse such abstract systems, architecture modelling languages are needed that provide architectural modelling concepts but still cater for formal analysis.

A last point, which is almost completely neglected in [8]—but which is considered in *Seven More Myths of Formal Methods* [3]—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: no learning without quick feedback, no quick feedback without tools.

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 SPIN tool was very successful: everybody can get the tool from the Web [17], 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, fun, and stimulates the use and appreciation 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; engineers found the use of Z sometimes frustrating, because "you couldn't do anything with the specifications".

Also during the actual development of the formal TD the tools which were used, SPIN [17] and ZTC [13], 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 non-existent.

Acknowledgments

The authors would like to thank Eric Burgers, Wouter Geurts, Franc Buve, Rijn Buve, Sjaak de Graaf, Hedde van de Lugt, Peter Bosman, Peter van de Heuvel and Robin Rijkers, all of CMG Public Sector B.V. in The Hague, for their active and enthusiastic participation during the interviews that form the basis of this paper. Ed Brinksma, Pim Kars, Wil Janssen, Job Zwiers and Theo Ruys from the University of Twente gave support and feedback during different phases of the BOS development. 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 and the anonymous referees are thanked for their constructive criticism.

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, SEI Series in Software Engineering*, Addison Wesley, Amsterdam, 1998.
2. A. Belinfante, J. Feenstra, R. 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 (Eds.), *Int. Workshop on Testing of Communicating Systems 12*, Kluwer Acadmic Publishers, Norwell, MA, 1999, pp. 179–196.

3. J. Bowen and M. Hinchey, "Seven more myths of formal methods," *IEEE Software*, Vol. 12, No. 4, pp. 34–41, 1995.
4. Cap Volmac (Cap Gemini Sogeti group), "SDW—System development workbench," 1993.
5. 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 J. Wing, J. Woodcock, and J. Davies (Eds.), *FM'99—World Congress on Formal Methods in the Development of Computing Systems II, Vol. 1709 of Lecture Notes in Computer Science*, 1999, pp. 1511–1526.
6. Dutch Ministry of Transport, Public Works and Water Management, "New waterway storm surge barrier—Innovative final element of delta project," URL: `http://www.minvenw.nl/rws/projects/svk/engels/index.html`.
7. W. Geurts, K. Wijbrans, and J. Tretmans, "Testing and formal methods—BOS project case Study," in *EuroSTAR'98: 6th European Int. Conference on Software Testing, Analysis & Review*, Munich, Germany, 1998, pp. 215–229.
8. A. Hall, "Seven myths of formal methods," *IEEE Software*, Vol. 6, No. 9, pp. 11–19, 1990.
9. D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1990.
10. G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1991.
11. G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.
12. IEC, *Functional Safety: Safety Related Systems, International Standard IEC 61508*, International Electrotechnical Commission, Geneva, Switzerland, 1996.
13. X. Jia, "ZTC: A Type Checker for Z—User's Guide," DePaul University, Chicago, USA, 1995.
14. P. Kars, "The application of PROMELA and SPIN in the BOS project," in J.-C. Grégoire, G. Holzmann, and D. Peled (Eds.), *The* SPIN *Verification System: The Second Workshop on the* SPIN *Verification System; Proceedings of a* DIMACS *Workshop, August 5, 1996*, Vol. 32 of DIMACS *Series in Discrete Mathematics and Theoretical Computer Science*, 1997, pp. 51–63.
15. P. Kars, "Formal methods in the design of a storm surge barrier control system," in G. Rozenberg and F. Vaandrager (Eds.), *Lectures on Embedded Systems, Vol. 1494 of Lecture Notes in Computer Science*, 1998, pp. 353–367.
16. J. Martin and P. Welch, "A design strategy for deadlock-free concurrent systems," *Transputer Communications* Vol. 3, No. 4, pp. 215–232, 1997.
17. Spin, "On-the-Fly, LTL Model Checking with SPIN," URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
18. J. Spivey, *The Z Notation: a Reference Manual (2nd edition)*, Prentice Hall, New York, 1992.
19. J. Tretmans, K. Wijbrans, and M. Chaudron, "Software engineering with formal methods: The development of a storm surge barrier control system—Seven myths of formal methods revisited," in S. Gnesi and D. Latella (Eds.), *Fourth Int. ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'99)—Proceedings of the FLoC Workshop*, Pisa, Italy, 1999, Vol. II, pp. 225–237. ISBN 88-7958-009-4.
20. R. de Vries and J. Tretmans, "On-the-fly conformance testing using SPIN," *Software Tools for Technology Transfer*, Vol. 2, No. 4, pp. 382–393, 2000.
21. P. Ward and S. Mellor, *Structured Development for Real-Time Systems, Yourdon Press Computing Series, Introduction & Tools*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985, Vol. 1.
22. K. Wijbrans, F. Buve, and W. Geurts, "Practical experiences in the BOS project," in *Proceedings of the Embedded Systems Symposium*, Eindhoven, The Netherlands, 1998.
23. K. Wijbrans and R. Buve, "Software bestuurt de stormvloedkering," *Software Release Magazine*, Vol. 50, No. 5, 1998. In Dutch.