Testing and Formal Methods
BOS Project Case Study

Wouter Geurts, Klaas Wijbrans
CMG Den Haag B.V.
Division Advanced Technology
P.O. Box 187
2501 CD Den Haag
The Netherlands
phone: +31 70 3029302
fax: +31 70 3029300
email: Wouter.Geurts@cmg.nl, Klaas.Wijbrans@cmg.nl

Jan Tretmans
University of Twente
Faculty of Computer Science
Formal Methods and Tools group
P.O. Box 217
7500 AE Enschede
The Netherlands
phone: +31 53 489 4287
fax: +31 53 489 3247
email: tretmans@cs.utwente.nl

May 25, 1999

Abstract

In this paper a case study is presented on the use of formal methods in the testing of a realistic (mission critical) system. The top level design was written in the formal language PROMELA, which made it possible to simulate a model of the system before it was built. Design flaws that otherwise only appear in integration testing were now visible before further design continued. Once a sound framework was established a more detailed description in the formal language Z was made. This design method made it possible that independent testers could perform high level structural tests on the individual modules, thereby providing a more detailed quality statement of the modules. These tests were systematically derived from de formal specifications. The increased effort in developing the formal description paid off in integration and system testing. In these phases no errors were found that lead to high level design modifications.

1 Introduction

1.1 The Storm Surge Barrier

The Netherlands are located in a low delta by 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

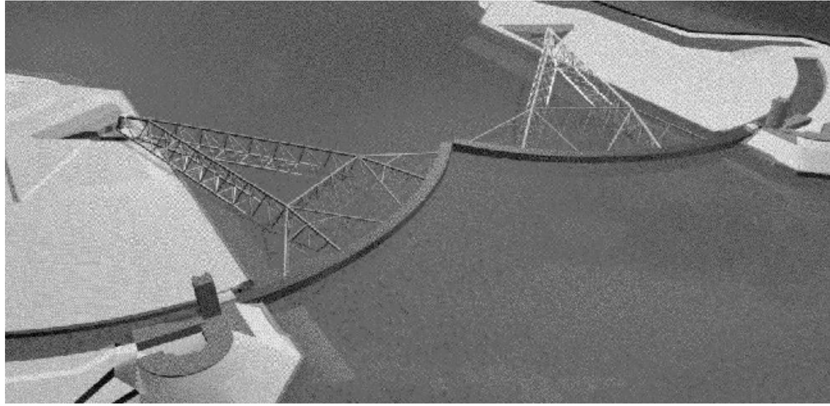

Figure 1: Computer drawing of the *Maeslant Kering* near Hoek van Holland (seen from the sea-side). The width of the waterway is about 300m.

yet again that the country was not safe. 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 was a defense 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 realisation of the Delta plan started soon after 1953 and in 1986 the impressive dam network in Zeeland was finished. The weak point in the defense was now the Nieuwe Waterweg. The Nieuwe Waterweg connects the main port of Rotterdam with the North Sea, hence it is a major shipping route, is and at the same time, being completely open, a major risk for flooding of Rotterdam, since major 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 fig. 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 lead to a 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, so the doors start to 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 any interference with the ship traffic.

The main requirement on the barrier is that is as reliable as a dike. Careful failure analysis showed that a 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.

1.2 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. But 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 guilder 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* (the Dutch Ministry of Transport, Public Works and Water Management), 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.

1.3 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 seen as 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 was 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".

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 [Hol91]. Z is a formal language based on set theory and predicate logic [Spi92].

None of the "highly recommended" techniques can assure completely the required safety, reliability and correctness [Bro95]. Only a carefully chosen combination of appropriate techniques can help to increase the confidence that the system has the required quality. This has lead 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. Independent test teams start the testing process as soon as the first specifications are available with reviewing these specifications, checking them for testability and preparing the test environment.

1.4 Paper Overview

This paper describes how validation of the BOS system was conducted, with particular emphasis on formal methods and their use in the testing process. Section 2 describes the Validation &

Verification activities that fit into a integral approach of risk based design and development. Section 3 elaborates on simulation and validation based on formal methods and on review and inspection techniques. For both, the influence they can have on the testing process are briefly discussed. Section 4 concentrates on the testing approach for BOS and the use of formal methods therein: the main benefits that were achieved and the perceived disadvantages of a testing approach based on formal specifications. Finally, section 5 sums up the main conclusions.

2 Validation and Verification

The general strategy of early risk identification and subsequent reduction according to IEC1508 is iterative. Starting with the risks for the users (in our case the risk of the flooding of Rotterdam, the risk of financial loss due to blocking of the Nieuwe Waterweg, and the risk of destruction of the sector doors due to higher water pressure from inside) new risks may emerge from choices in the design process, and later from choices in the implementation process. It is crucial that these choices are assessed with respect to their impact on the test process, e.g., the identification of subsystems and libraries determines the integration procedure of the system, which may cause an unnecessary high pressure on the test team (like in big bang integration testing).

This general strategy is concretized in the Validation and Verification plan (V&V-plan). This V&V-plan combines several quality assessment techniques in such a way that they enhance each other. Testing can be used more direct if some aspects of the system have been verified already in reviews or inspections. An example of enhancement of complementary validation techniques is the following. Consider a system with two (non-interacting) subsystems each having 10 states. The number of states of the global system is 100. Without further knowledge a tester would have to test the behaviour in all these states. If the two subsystems are designed (and coded) independently, and this independency is validated using reviews, the test effort can be reduced drastically: each subsystem can be tested independently, while the correctness of their integration is deducted from assumptions already checked by reviews. So if testers can use reviews, the early test preparations can focus on the removal of risks, or reducing the effort in the testing process. There is much profit to be gained in these kind of combinations of different software validation techniques.

The top level structure of BOS and the internal process-scheduling mechanism have been modelled in PROMELA an example of this is given in fig. 2 where the barrier controling process `KEzWWBBestWaterweg` is given. The other processes (including an external world process) have also been modeled. In section 3 the approach of the validation and verification of this structure is given. In fig. 3 a typical Z-schema is given, that is 'called' from the 'PROMELA program'. The schema is a logical combination of subschemas. The verification approach to the Z-schemas is described in section 3, while the advantages of using these schemas to derive test cases are discussed in section 4.

3 Validation of the Design

The quality of the design of a system is crucial for the number of problems encountered during integration and system testing and for its performance in the field. Many projects encounter specific problems only late in the development life cycle. The common denominator of these problems is that modules which seem to function correctly individually cause, when combined, undesired system behavior and unexpected or unreproducable errors. This kind of problems can be prevented by structuring the design in such a way that these problems can never be introduced or that they can at least be easily detected.

Within BOS, the following risks were identified at the design level:

- Concurrency within the system itself. Especially because BOS consists of multiple communicating processes, a risk of either deadlock or bad data due to synchronization problems is present.

```
proc KEpWWBBestWaterweg (
  chan IBcPMACmnd              /* in:  cmnds from Process Manager */
  chan IBcPMAResp              /* out: responses to Process Manager */
  chan PScWWBCmnd              /* in:  cmnds from script interpreter */
  chan PScWWBResp              /* out: responses to script interpreter */
  chan IBctsTrigger            /* in:  timer events */
  chan GUcTrigger              /* out: triggers to GUI */
  chan KEcWWICmndList          /* out: cmnds to interface processes */
  chan KEcWWIResponses         /* in:  responses from interface processes */
) {
  IBcPMAResp ! ipsINITIALISED;
  if
  :: IBcPMACmnd ? ibcSTOP -> skip;
  :: IBcPMACmnd ? ibcSTART ->
     [...] (some initialisation)
     do
     :: IBcPMACmnd ? ibcSTOP -> break;

     :: IBctsTrigger ? KEgtdWWBCheckOffset ( mod KEgtdWWBCheckPeriod ) ->
        KEzWWBPrimTimeout;
        KEzWWBHartbeatTimeout;
        [...]

     :: PScWWBCmnd ? KEsWWBSendCommand ->
        if
        :: KEzWWBBarrierCmndErr
        :: KEzWWBBarrierCmndOk -> KEzWWBSendActData
        fi
     [...]

     od
  fi
}
```

Figure 2: Promela Example. This example describes the barrier controlling process `KEpWWBBest-Waterweg`. The process will react on start/stop commands from the (central) process manager, on time triggers and on commands of the total functional controlling process called the 'script interpreter'.

$KEzWWBHartbeatTimeout ==$
$\quad WWBHartbeatTimeoutOperational \lor$
$\quad WWBHartbeatTimeoutFailed \lor$
$\quad WWBHartbeatTimeoutIgnore$

[...]

WWBHartbeatTimeoutOperational

$now? : IBctsTrigger;$
$\Xi SKEvaWWBConfig;$
$\Delta SKEvWWBMode;$
$\Delta SDBvdTrigData;$
$\Delta SKEvWWBCmnd;$
$\Delta SKEvWWBBesRegs;$
$\Xi SKEvWWBHartbeat;$
$\Delta SKEvWWBPrimair;$
$trigger! : GUcTrigger;$
$alert! : IBcAlertChannel;$

$now? \bmod KEgtdWWBCheckPeriod = KEgtdWWBCheckOffset;$
$now? \geq KEgtsWWBLastHartbeat + KEgtdWWBHartbeatTimeout;$
$KEgeWWBControlMode = bsmMAINTAINANCE;$
$KEgeWWBInterfaceStatus = ifsOPERATIONAL;$

$KEgeWWBInterfaceStatus' = ifsFAILED;$
$alert! = \{KE_sWWBCommFailureNoHartbeat(KEgtsWWBLastHartbeat);$
$KEgfWWBBesDataValid = wbdBESWDATAVALID \Rightarrow$
$\quad KEzWWBSetStateUNDEFINED;$
$KEgfWWBBesDataGeldig' = wbdBESWDATANOTVALID;$

Figure 3: Detail of one of the Z-schemas in the PROMELA specification of fig. 2.

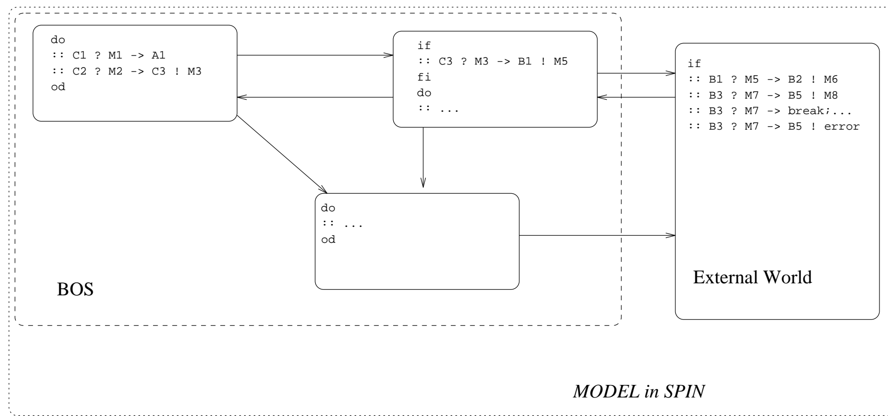

Figure 4: The individual processes as well as the outside world have been modelled in PROMELA.

- Incomplete specification of behavior. In many projects the design is incomplete with respect to the overall behavior of a process. Only the behavior when all functions within a system perform correctly and no faults occur in the system environment is specified. All error handling and recovery is left to the programmer.
- Interface faults. Often, interface protocols are not specified completely, or their specification is not validated for interface robustness.
- Miscommunication and misinterpretation between designers, testers and implementers. Many faults are introduced in systems due to a lack of understanding between these groups.

To ensure the desired quality and to mitigate these risks, measures must be introduced in the design process. These measures either prevent the occurrence of specific classes of errors, or ensure their early detection. Within the BOS project, the following measures were taken at the design level: a formal strategy for deadlock free design, validation of design concepts and interface protocols and design reviews in combination with multiple formal notations.

3.1 Formal Strategy for Deadlock Free Design

A common problem in concurrent systems is the occurrence of deadlocks while the system is running. These deadlocks may result in a kind of domino effect in which the whole system tumbles over. Sometimes, designers try to prevent these deadlocks by using shared data structures without guards on the validity of this data. Though this strategy prevents deadlocks from occurring, often synchronization problems arise, in which processes simply use incorrect data with the same disastrous results for the system behavior.

It is, however, possible to design systems that are guaranteed to be free from deadlocks. In those cases the designer should limit himself in the constructions used. Within BOS, the client-server paradigm from [MW97] was applied. The success of this approach is evident: in the concurrent system that BOS is, no problems due to either deadlocks or bad synchronization have appeared during integration testing, system testing and its operation until now. For further information on this approach, the reader is referred to the article mentioned.

3.2 Validation of Design Concepts and Interface Protocols

A second major risk is the improper use of design concepts for starting and stopping the system, error recovery and for incompletely or incorrectly specified interface protocols. Within the BOS project, this risk was mitigated by using the PROMELA protocol modeling language in combination with the SPIN tool for validating these concepts and protocols.

The BOS system model was constructed with an iterative procedure. First a pessimistic model of the external world was constructed. In this model the outside world regularly fails to respond, corrupts data etc. Within this framework of a unreliable environment the model for the BOS system was tested on robustness. All design measures to handle rare situations were tested using simulation and state space analysis (using the tool Spin). Any suspicious behavior of the model lead to adjustments and revalidation. Finally, when no adjustments were necessary anymore, the design was, due to its structure, capable of performing its primary tasks despite of the fact that some processes (or even some subsystems) might have crashed or are too busy. In the final design the simplest alternative behaves well under stress has been adopted. The iterative way of working guarantees that the final design is fully validated.

Similar to the BOS internals, the interfaces between BOS and the outside world were validated. Several used protocols appeared to have the danger of providing incorrect data when under stress. Using the Promela model of the protocol and the BOS design, safe alternatives have been explored. This approach paid off: no problems with the validated protocols have been found later in the project.

3.3 Design Reviews in Combination with Multiple Formal Notations

Reviews are necessary and have proven to be efficient. In the BOS design the simultaneous use of different descriptions (Promela, Z and natural language) proved to be very helpfull in pointing out errors. The Z description as in fig. 3 is accompanied by a description in natural language. The formal languages helped to assess properties like completenes, while the natural language helped in the communication with, e.g., the contractor. The process of checking statically is most helped by a continuous learning organisation. This means that someone should check the possible prevention of every problem. A yes to the question 'Could we have prevented this problem by better review check lists?' should lead to these better check lists. So the important point with introducing these reviews is that feed back on the review focus should be based on problems found.

Reviews are best done by all people affected by a document or product. Both implementors and testers should review the design, eventually suggesting alterative design constructions. It is not a problem that focus areas of different reviewers overlap. Testers should focus on the aspects of the design affecting the test:

Testability (observability and controlability) This is a point that can really save tight schedules. The ultimate goal is a well-tested system. If the system has been designed such that the internal states are not well visible to a tester or the system can not be brought to a specific state, a tester will have a hard time to first compose complete test cases and when a problem is found, the implementer *and* designer will have a hard time to prove or disprove the validity of the problem. Generally, imple design modifications can save months of work.

Completeness of Design (coverage of all possible stimuli and all possible states of the system). This is a typical point which should be assessed by all design reviewers. Good designing practice prescribes completeness. Check it anyway, it is little work extra when one is heading for complete test specifications. In the example of fig. 3 the complete coverage of preconditions can be checked by writing out the precondition of the total schema `KEzWWBHartbeatTimeout` in terms of the subschemas. Simple mathematics can prove the total coverage of preconditions (the precondition of `KEzWWBHartbeatTimeout` should be *true*).

Completeness of Interface Specifications An interface should be specified in a complete and unambiguous way. We found that the *process* of trying to model the interface in a language like Promela alone is revealing so much ambiguous aspects of the specification that this is recommended for each interface. Such a model can guide a tester to a full test of critical scenarios.

Standard of Exception/Failure Handling Most designers treat (minor) exception and failure handling as an implementation issue. In mission critical systems one should be aware of single

points of failure (one could think of the AT&T long-distance network collapse or the software failure in the Ariane 5). In the integral risk based approach for mission critical systems of section 2 some exceptions have been identified as risky. Testing should focus on the behavior of the system exposed to these exceptions. It is good practice to accompany the design with a standard for handling of minor exceptions (what to do when a device is not ready, connection time out etc.).

4 Testing with Formal Methods

4.1 Approach

All possible stimuli of a module are described in a structured manner in the PROMELA part of the specification, e.g., see figure 2. This constitutes the basis for systematic test suite derivation for module testing: for each possible stimulus at least one test case is devised. Moreover, to each possible PROMELA stimulus there are Z-schemas attached which describe the conditions which should hold before and after execution of the PROMELA action. Test cases are subsequently devised for all possible combinations (i.e., disjunctive normal form) of preconditions of all Z-schemas. This test generation process was performed manually, but (partial) automation could be envisaged, see also section 4.5.

A test case devised in this way usually has a structure comprised of three parts. In the first part of the test case the system is brought into a state where the specified stimulus can be applied (the *test preamble*). The second part consists of applying the stimulus and checking the results as specified in the appropriate Z-schema (the *test body*). In the final part of the test case the system is transferred to a well-defined state, e.g., the initial state, in order to be able to apply the next test case (the *test post-amble*). The structure of a test suite is provided by the structure of the design.

Consider the Z-schema `WWBHartbeatTimeoutOperational` of figure 3. A test case for this schema should first check whether this schema is applicable, i.e., whether the conditions of the schema apply (the third and fourth constraint of the schema). Moreover, the seventh line contains an implication which can be satisfied or not; this leads to two test cases, one where `KEgfWWB-BesDataValid=wbdBESWDATANOTVALID` and one where `KEgfWWBBesDataValid = wbdBESWDATA-VALID`. Then the test case should wait until both timing constraints (first and second constraint) are satisfied. Then the result of the schema, i.e., the output of `alert!`, should be checked. In a pseudo language one of the two test cases could be described by:

```
TEST CASE WWBHartbeatTimeoutOperational_1
   CheckCondition( KEgeWWBInterfaceStatus == ifsOPERATIONAL )
   CheckCondition( KEgeWWBControlMode == bsmMAINTAINANCE )
   CheckCondition( KEgfWWBBesDataValid == wbdBESWDATANOTVALID )
   WaitTil( KEgtsWWBLastHartbeat + KEgtdWWBHartbeatTimeout )
   WaitTil( KEgtdWWBCheckPeriod, KEgtdWWBCheckOffset )

   CheckAlert( KE_sWWBCommFailureNoHartbeat )
END TEST CASE
```

The other case is similar but with the precondition `KEgfWWBBesDataValid = wbdBESWDATA-VALID`, which also checks the additional execution of the schema `KEzWWBSetStateUNDEFINED`.

Test suite validation The test suite has been designed to cover the risks identified earlier in the project in the V&V-plan and in analysis of found problems. Several kinds of coverage can be used to assess the tests. One could think of coverage of requirements, coverage of agreed use cases, coverage of design statements, etc.

A well-defined test design (using requirements trace-ability and inspections) can implement coverage of requirements and use cases. Although this check is useful and sometimes necessary, it

is done manually, and therefore is costly. In our case the direct linkage of test cases to Z schemas guarantees a certain amount of coverage.

The design coverage can be measured in a derived but independent way by means of a code coverage tool. In principle no conclusions about design coverage can be drawn on the basis of code coverage analysis, but using strict code inspections in combination with coding standards strong indications can be obtained. A code coverage tool indicates which code was executed during test suite execution. Subsequently, code inspections should indicate which functionality from the design was not executed. There are some necessary items that should be described in the coding standards, viz. a one-to-one mapping of design items (buffers, functions, variables) and a veto on fancy code, prescribing standard (well-tested!) implementations of design description. Using the combination of disciplines a tool that measures only line coverage by instrumenting the code is already extremely useful.

Besides an assessment on the test suite, the code coverage analysis gives hints about missed code (denoting extra code or missed areas of the design). Ultimately, the code coverage analysis helps to improve the test design process.

4.2 Test Environment

Test execution can only start when the system is ready to run. If a well-defined test environment is designed, (high-level) tests can be written. The test environment is the machine that translates (high-level) test cases to (low-level) interactions with the system (i.e., messages, interrupts, database access, keyboard action, mouse actions, etc.). We have chosen to introduce the same communication abstractions that the application itself uses. In this way, a test language was defined that is as abstract as the Z-schemas in the design, meaning that the full test set based on the design can be written down when the design is ready. Once the implementation has been defined (precise implementation of messages, etc.) the high level vocabulary could be translated to this implementation and the tests are ready to run when the code is sent to test.

The big advantages of this approach for the test are less maintenance, higher flexibility and fast reproduction of problems. Additional advantages are that the test environment could be used as a glass-box test environment usable for the implementer tests.

The test environment has been implemented in PERL[WCS96] as a module. Test suites are automatically executed. Test cases are implemented as PERL programs using a process algebra sort of structure where the primitive test actions (send, receive, check etc.) can be combined to full tests with sequence, choice and parallel operators. The fact that the test language does not confine test cases to be pure sequences of test actions is a powerful feature of the test language.

4.3 Benefits of Formal Methods

The use of formal methods in specifying the design helped the testing process in several ways. In this section we elaborate on a few aspects.

Preciseness Preciseness and diminished ambiguity of the formal specifications were of much help during testing and (more) in test result analysis.

Usually, one of the main problems in software testing is the lack of a precise, complete, consistent, unambiguous, clear and structured specification which can be used as the basis for testing. This lack leads to difficulties during test generation, such as the question what are the exact requirements that should be tested, and during test analysis, where, in case of detected errors, this is often due to misinterpretations by either the implementers or the testers. Discussions about intended system functionality tend to proliferate into the testing process.

Use of formal specifications as the basis for both implementation and test generation helped solving these traditional problems. No long discussions with implementers or designers were needed for interpretation of requirements or unraveling of the intention of (a part of) a specification: the formal text served as a precise and consistent reference point. Altogether, efficiency

improved by clear and precise communication about the product, its intended functionality and its requirements.

Of course, this approach makes the specification and the design more important in the development trajectory and, consequently, thorough validation of the specification and of the design are of prime importance, see section 3.

Planning The structured and systematic way of developing test scripts, together with the fact that measures can be applied to Z specifications (in its simplest form the size of the Z specification in terms of lines of Z-code can serve as a measure), allowed to estimate and to plan the test process. The estimates and plans were quite precise and successful. This was particularly true for the later modules, after some experience had been obtained and metrics had been collected from the first modules.

Moreover, due to completeness and preciseness of the formal specifications, less design decisions were autonomously made by implementers, and this resulted in less unexpected surprises detected during testing, which in traditional projects often hamper the planning.

Concurrent development Due to the fact that implementers had less freedom in making independent design decisions, the testing process could really start immediately after the design was completed, hence saving overall project time. Test suite specification was usually finished before or at the same time as code implementation, so that test execution could immediately start when the code was ready.

Structure A Z specification is structured and consists of (formally specified) requirements that each system operation must fulfil together with a pre- and post-condition. Both tester and implementer benefit from this structure, adopting a similar structure for the tests and code, respectively. This saves time during testing. Developing a test suite consists of systematically developing a test case for each precondition. Although, within the BOS project, this process of test script development was manual, it was structured, systematic and efficient, and it resulted in a well-structured test suite.

Thoroughness and completeness The thoroughness and completeness of testing increased since a formal specification is more detailed and more complete than a usual functional specification. In this sense, testing based on Z specifications shifts somewhat from functional towards structural testing, while keeping the advantages (independency of functional testing and enhanced visibility of structural testing).

An effective check on the completeness and abstraction level of the design appeared to be the construction of test specifications based only on the design. If this turns out to be impossible, then valuable time will be lost by the implementer making design decisions, hence adaptation of the design is preferred.

Code coverage The test suites were developed aiming at 'Z schema condition coverage'. This strategy turned out to be quite successful. The quality of the obtained test suites measured in terms of line code coverage turned out to be quite high: $> 90\%$ coverage was usually obtained for the first test set developed. The missed code could easily be identified as exception handling of illegal states, i.e., should never be executed in a correctly functioning system. It should be noted that this high code coverage is partly due to the quality of the test sets and partly to the fact that the same specifications which were the basis for test set development, also were the basis for code implementation, where the structure of the formal specification was usually preserved as much as possible (cf. the previous remark on shifting from functional towards structural testing).

4.4 Disadvantages of Formal Methods

Of course, some disadvantages are related to testing based on formal methods. Actually, these disadvantages are not as much related to testing, but more to the use of formal methods in general.

Education The most important problem which was encountered was the lack of formal methods expertise and trained personnel. This implied that education and training were necessary in the BOS project. Courses in formal methods were set up and a collaboration with a university group (The Formal Methods and Tools group of the University of Twente) was established. However, while formal languages can be learned in courses (they are hardly more difficult than high-level programming languages), it is more difficult to get real experience, develop pragmatics and heuristics, and find ways of embedding formal methods in existing development trajectories. All this requires experience, skill and a lot of practice. This is more difficult to acquire during a single project, and also university knowledge in formal methods is not of much help for this.

What can be observed in the BOS project is that this kind of knowledge was gradually acquired during the project. In the first modules which were developed the use of formal methods was really considered to be a burden. In modules which were developed later on in the project, when pragmatics and styles of formal specification had been established, the benefits of the use of formal methods were certainly apparent. It is expected that in future projects, when this kind of knowledge is common practice, the advantages will even more prevail.

Planning A second point of concern is overall project planning. When using formal methods there will be more time and effort invested in specification and design. Coding and testing will start later in the project compared with traditional project planning. This should not worry (project) managers. The extra time spent during specification and design is usually gained during implementation and in particular, as the BOS experience showed, during testing.

Tools For Promela the tool Spin was used for checking the design, both statically and dynamically. In particular, simulation and verification are powerful techniques to analyse Promela models, however, models should not be too complex to be handled by Spin.

For Z only static checking tools are available. Although such tools are of great help in making the specification statically complete (all items have been defined, etc.), additional tools which could help in checking preconditions, etc. would be of great help. Such tools exist, but they can only deal with very small toy specifications and not with the schemas of BOS.

Organization Finally, of course, formal methods are not a silver bullet. It is just a technique to improve the quality of the resulting product, mainly by increasing preciseness. It should be applied with care in a well organized and solid project: formalized chaos is even worse than non-formal chaos.

4.5 Future Testing

Testing is, and will stay, an important technique to check and improve the quality of software products. However, often testing is a separate, disconnected task at the end of the project development cycle. To be really effective, testing should be an integrated and integral part the quality assurance process, and the testing process should start immediately at the launch of the project.

Only if testing is an integral part of system quality assurance, testing can be directed by risk analysis – test those system aspects thoroughly which are safety critical or risky –, and can be complemented with other validation techniques – certain system aspects are tested, others are (formally) verified, and yet others are more efficiently checked by reviewing. It is our belief that a careful combination of appropriate validation techniques, which are applied based on thorough risk and failure analysis, is necessary to assure the quality of future generation safety critical systems.

With respect to the techniques of testing based on formal methods, future developments will be in the direction of automation. This concerns both automation of test derivation from formal

specifications and automatic test implementation, test execution and test analysis. The latter is especially important for regression testing. Only if regression testing can be fully automated it will be really performed after each (minor) modification of the system. System modifications and adaptations are usually very dangerous for system correctness and system consistency.

Automatic test derivation from formal specifications is currently an important topic of academic research. First academic prototype tools working with formal languages like SDL, Lotos and Promela are now available [FJJV96, STW96, PS97, SKGH97, VT98]. It is expected that automatic test derivation will give an important boost to the use of formal methods: investments in using formal methods will soon pay off, if your test suite is for free once you have developed a formal specification.

5 Conclusions

The approach for developing the mission critical system BOS, consisting of risk based analysis, careful combination of complementary validation techniques, the use of formal methods, independent testing based on the formal specifications and a strict and well organized project development structure, can be considered successful. The system was delivered in time without major software bugs.

The approach of testing based on formal design specifications contributed to the overall quality of the product. The preciseness and consistency of formal specifications is of great help in the testing process. Planning, thoroughness, completeness, structure, accuracy and cost of testing are better than for comparable projects. The investments to achieve these improvements are that more effort has to be spent in the design phase of the product to develop the formal descriptions, and that knowledge about formal methods, and the way how to work with them in large projects, has to be acquired. But the overall balance is certainly in favour of the use of formal methods, and for future, analogous projects CMG would advise to use formal methods. For future projects the gain could even be greater since the knowledge of using formal methods is now available at CMG. It should be noted, however, that the use of formal methods within a project only makes sense if the project is well structured and well organized, and if basic software engineering practices, like configuration management, design and code reviews, etc. are established.

An aspect which needs attention is the availability of tools for formal methods. Available tools are not powerful enough to cope with the large formal specifications of the BOS project, and where they do cope with them, their functionality is very restricted. Development of more mature tools is necessary, and in particular in the testing arena, expectations are high with respect to tools for the automatic derivations of tests from formal specifications.

References

[Bro95] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary ed. edition, 1995.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification CAV'96*. Lecture Notes in Computer Science 1102, Springer-Verlag, 1996.

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

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

[MW97] J.M.R. Martin and P.H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4), June 1997.

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

[SKGH97] M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. – Autolink – a tool for the automatic and semi-automatic test generation. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme*, number Nr. 315 in GMD-Studien, St. Augustin, 1997. GI/ITG-Fachgespräch, GMD.

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

[STW96] Dutch Technology Foundation STW. *Côte de Resyste* – COnformance TEsting of REactive SYSTEms. Project proposal STW TIF.4111, University of Twente, Eindhoven University of Technology, Philips Research Laboratories, KPN Research, Utrecht, The Netherlands, 1996.

[VT98] R.G. Vries and J. Tretmans. On-the-fly conformance testing using SPIN. In *Fourth SPIN Workshop*, 1998.

[WCS96] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, second ed. edition, 1996.